NeXTSTEP™ OPERATING SYSTEM SOFTWARE

NeXTSTEP Developer's Library
NeXT Computer, Inc.

Release 3

Addison-Wesley Publishing Company
Reading, Massachusetts · Menlo Park, California · New York · Don Mills, Ontario
Wokingham, England · Amsterdam · Bonn · Sydney · Singapore · Tokyo · Madrid
San Juan · Paris · Seoul · Milan · Mexico City · Taipei

NeXT and the publishers have tried to make the information contained in this manual as accurate and reliable as possible, but assume no responsibility for errors or omissions. They disclaim any warranty of any kind, whether express or implied, as to any matter whatsoever relating to this manual, including without limitation the merchantability or fitness for any particular purpose. In no event shall NeXT or the publishers be liable for any indirect, special, incidental, or consequential damages arising out of purchase or use of this manual or the information contained herein. NeXT will from time to time revise the software described in this manual and reserves the right to make such changes without obligation to notify the purchaser.

NeXTSTEP Operating System Software Copyright © 1990–1992 by NeXT Computer, Inc.
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher or copyright owner. Printed in the United States of America. Published simultaneously in Canada.

NeXTSTEP 3.0 Copyright © 1988–1992 by NeXT Computer, Inc. All rights reserved. Certain portions of the software are copyrighted by third parties. U.S. Pat. No. 4,982,343. Other Patents Pending.

NeXT, the NeXT logo, NeXTSTEP, Application Kit, BusProbe, Digital Librarian, NBIC, NeXTbus, NeXTcube, NeXTdimension, NeXTstation, Sound Kit, and Workspace Manager are trademarks of NeXT Computer, Inc. Display PostScript and PostScript are registered trademarks of Adobe Systems Incorporated. NFS is a registered trademark of Sun Microsystems, Inc. UNIX is a registered trademark of UNIX Systems Laboratories, Inc. All other trademarks mentioned belong to their respective owners.

Restricted Rights Legend: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 [or, if applicable, similar clauses at FAR 52.227-19 or NASA FAR Supp. 52.227-86].

This manual describes NeXTSTEP Release 3.

Written by NeXT Publications.

This manual was designed, written, and produced on NeXT computers. Proofs were printed on a NeXT 400 dpi Laser Printer and NeXT Color Printer. Final pages were transferred directly from a NeXT optical disk to film using NeXT computers and an electronic imagesetter.

1 2 3 4 5 7 8 9 -AL-9695949392
First printing, September 1992

ISBN 0-201-63252-7

Contents

Introduction

Part 1: The Mach Operating System

1-1 Chapter 1: Mach Concepts
1-2 Design Philosophy
1-4 The Mach Kernel
1-26 Mach C-Thread Functions
1-32 Mach Exception Handling

2-1 Chapter 2: Using Mach Messages
2-2 Message Structure
2-4 Creating Messages by Hand
2-8 Mach Interface Generator

3-1 Chapter 3: Using Loadable Kernel Servers
3-2 How Loadable Kernel Servers Work
3-3 Overview of Kernel-Server Loader Functions
3-4 Functions for Asynchronous Messages

4-1 Chapter 4: Mach Functions
4-2 C-Thread Functions
4-22 Mach Kernel Functions
4-115 Bootstrap Server Functions
4-126 Network Name Server Functions
4-129 Kernel-Server Loader Functions

Part 2: Writing Loadable Kernel Servers

5-1 Chapter 5: Overview of Loadable Kernel Servers
5-1 Advantages and Disadvantages of Loadable Kernel Servers
5-2 Supported Types of Loadable Kernel Servers
5-3 How Loadable Kernel Servers Work
5-5 Before You Start

6-1 Chapter 6: Designing Loadable Kernel Servers
6-1 Choosing an Interface for Your Server
6-2 Tips for Writing Server Code
6-4 Building In Debugging Code
6-6 Kernel-Server Loader Requirements
6-7 Considerations for Message-Based Servers
6-13 Considerations for UNIX-Based Servers

7-1 Chapter 7: NeXTbus Device Drivers
7-2 NeXTbus Address Space
7-4 Writing Code to Control the Hardware
7-8 NeXTbus Board Registers

8-1 Chapter 8: Network Modules
8-2 Overview
8-3 Network Objects
8-4 Functions Implemented in the Network Module
8-12 Notes for Specific Interfaces

9-1 Chapter 9: Building, Loading, and Debugging Loadable Kernel Servers
9-1 Building a Loadable Kernel Server
9-3 Loading a Server into the Kernel
9-4 Tools for Debugging Servers
9-7 Debugging Servers with GDB

10-1 Chapter 10: Kernel Support Functions
10-2 General Functions
10-54 Network Functions

A-1 Appendix A: Utilities for Loadable Kernel Servers
A-2 Kernel-Server Loader (**kern_loader**)
A-4 Kernel-Server Linker (**kl_ld**)
A-8 Kernel-Server Utility (**kl_util**)
A-10 Kernel-Server Log Command (**kl_log**)

B-1 Appendix B: The ROM Monitor and NMI Mini-Monitor
B-1 ROM Monitor Commands
B-6 NMI Mini-Monitor Commands

C-1 Appendix C: Summary of Kernel Support Functions
C-1 General Functions
C-6 Network Functions

Suggested Reading

Glossary

Index

Introduction

This manual describes the NeXT™ Mach operating system. It's part of a collection of manuals called the *NeXTSTEP Developer's Library*, which is published by Addison-Wesley.

You don't have to read this manual to be able to create NeXTSTEP™ applications. However, the information in this manual is necessary if you want to use Mach features such as multiple threads in a task, or if you want to write add modules such as device drivers to the operating system.

UNIX®-related topics aren't covered in this manual. For information about the UNIX operating system, you should refer to the books listed in "Suggested Reading" (at the end of this manual) or to the on-line UNIX manual pages (available through Digital Librarian™).

A version of this manual is stored on-line, and is accessible in Digital Librarian through the bookshelf **/NextLibrary/Bookshelves/NextDeveloper.bshlf**. The bookshelf also contains Release Notes, which provide last-minute information about the latest release of the software.

How This Manual Is Organized

This manual is divided into two parts.

- Part 1 discusses the NeXT Mach operating system, in general. The information in it can be used when writing both user-level programs (such as NeXTSTEP applications) and kernel modules.
- Part 2 discusses how to create *loadable kernel servers*. Loadable kernel servers are software modules that are loaded into the kernel after it has booted. They're used to implement kernel functionality such as device drivers.

Conventions

Syntax Notation

Where this manual shows the syntax of a function, command, or other programming element, the use of bold, italic, square brackets, and ellipsis has special significance, as described here.

Bold denotes words or characters that are to be taken literally (typed as they appear). *Italic* denotes words that represent something else or can be varied. For example, the syntax

print *expression*

means that you follow the word **print** with an expression.

Square brackets [ ] mean that the enclosed syntax is optional, except when they're bold **[ ]**, in which case they're to be taken literally. The exceptions are few and will be clear from the context. For example,

pointer [*filename*]

means that you type a pointer with or without a file name after it, but

[*receiver message***]**

means that you specify a receiver and a message enclosed in square brackets.

Ellipsis (...) indicates that the previous syntax element may be repeated. For example:

Syntax	Allows
pointer ...	One or more pointers
pointer [, *pointer*] ...	One or more pointers separated by commas
pointer [*filename* ...]	A pointer optionally followed by one or more file names
pointer [, *filename*] ...	A pointer optionally followed by a comma and one or more file names separated by commas

Notes and Warnings

Note: Paragraphs like this contain incidental information that may be of interest to curious readers but can safely be skipped.

Important: Paragraphs like this contain important information.

Warning: Paragraphs like this are extremely important to read.

Note

Gray boxes like this contain information that isn't necessary for understanding the information discussed nearby, but that is useful when you start to use the information.

Part 1
The Mach Operating System

Contents of Part 1

1-1 Chapter 1: Mach Concepts
1-2 Design Philosophy
1-4 The Mach Kernel
1-5 Mach Tasks and Threads
1-6 Task and Thread Ports
1-8 Mach Ports and Messages
1-9 Port Access Rights
1-10 Port Sets
1-10 Port Names
1-11 Port Queues
1-11 Extended Communication Functionality
1-12 Messaging in a Network Environment
1-13 Mach Virtual Memory Management
1-13 Demand Paging
1-14 Inheritance and Protection of Memory
1-14 Inheritance
1-15 Protection
1-15 Interprocess Communication
1-16 Memory-Mapped Files
1-16 Paging Objects
1-16 Virtual Memory Functions
1-17 Program Examples: Virtual Memory
1-23 Mach Scheduling
1-24 Priorities
1-25 Policies
1-26 Mach C-Thread Functions
1-26 Using External Functions and Methods

1-27 Using Shared Variables
1-28 Synchronization of Variables
1-30 Program Example: C Threads
1-32 Mach Exception Handling
1-33 The UNIX Approach to Exception Handling
1-34 A Model for Generalized Exception Handling
1-35 Exception Handling in Mach
1-36 Exception Ports
1-37 User Extensibility
1-38 Implementing Error Handlers
1-39 Implementing Debuggers
1-40 Debugger Attachment
1-40 Parallel and Distributed Debugging
1-41 GDB Enhancements
1-42 Exception Classification
1-43 Kernel Interface
1-44 Program Example: Exception Handling

2-1 Chapter 2: Using Mach Messages
2-2 Message Structure
2-2 Message Header
2-3 Message Body
2-4 Creating Messages by Hand
2-4 Setting Up a Simple Message
2-6 Setting Up a Nonsimple Message
2-7 Setting Up a Reply Message
2-8 Mach Interface Generator
2-9 Creating Mach Servers with MiG
2-10 Client's View
2-10 Common Error Codes
2-11 Out-of-Line Data
2-11 Compiling the Client
2-11 Programming Example

2-16 MiG Specification File
2-16 Subsystem Identification
2-17 Type Declarations
2-17 Simple Types
2-20 Structured Types
2-21 Pointer Types
2-21 Import Declarations
2-22 Operation Descriptions
2-24 Options Declarations
2-24 The **waittime** Specification
2-25 The **sendtime** Specification
2-25 The **msgtype** Specification
2-26 The **error** Specification
2-26 The **serverprefix** Specification
2-26 The **userprefix** Specification
2-27 The **rcsid** Specification
2-27 Syntax Summary
2-30 Compiling MiG Specification Files

3-1 Chapter 3: Using Loadable Kernel Servers
3-2 How Loadable Kernel Servers Work
3-3 Overview of Kernel-Server Loader Functions
3-4 Functions for Asynchronous Messages
3-4 Common Code for Handling Reply Messages
3-8 Handling a Status Message
3-9 Handling a Synchronization Message
3-9 Handling a Log Message

4-1 Chapter 4: Mach Functions
4-2 C-Thread Functions
4-22 Mach Kernel Functions
4-115 Bootstrap Server Functions
4-126 Network Name Server Functions
4-129 Kernel-Server Loader Functions

Introduction to Part 1

The following four chapters describe the NeXT Mach Operating System.

- Chapter 1, "Mach Concepts," describes NeXT's version of Mach. It discusses concepts such as the kernel, tasks and threads, and ports and messages. This chapter also explains how Mach manages virtual memory allocation and how it handles exceptions.

- Chapter 2, "Using Mach Messages," describes how to create Mach messages, either by hand or by using the Mach Interface Generator (MiG).

- Chapter 3, "Using Loadable Kernel Servers," describes how to communicate with loadable kernel servers and the kernel-server loader by using the kernel-server loader functions.

- Chapter 4, "Mach Functions," provides detailed descriptions of all Mach operating system functions that are available to user-level programs. Some of these functions are also available to loadable kernel servers. The functions described in Chapter 4 are summarized in the *NeXTSTEP Programming Interface Summary*.

Note: UNIX library functions and system calls aren't covered in Chapter 4; they're described in the on-line UNIX manual pages.

1 *Mach Concepts*

Mach, the operating system of all NeXT computers, was designed by researchers at Carnegie Mellon University (CMU). Mach is based on a simple communication-oriented kernel, and is designed to support distributed and parallel computation while still providing UNIX 4.3BSD compatibility.

The NeXT Mach operating system is a port of CMU Release 2.0, with additional features both from NeXT and from later versions of CMU Mach. NeXT-only features include the Bootstrap Server and loadable kernel servers. Features from CMU Release 2.5 and beyond include scheduling and some details of messaging.

Mach consists of the following components:

- A small, extensible system kernel that provides scheduling, virtual memory, and interprocess communications; the kernel exports a small number of abstractions to the user through an integrated interface.

- Operating system support environments that provide distributed file access, transparent network interprocess communication, remote execution facilities, and UNIX 4.3BSD emulation. Many traditional operating system functions can be implemented by user programs or servers outside the kernel.

Although Mach's design is conceptually unlike that of UNIX 4.3BSD, it maintains UNIX 4.3BSD compatibility. Mach system calls are upwardly compatible with those of UNIX 4.3BSD, and Mach supports UNIX 4.3BSD commands. This compatibility is transparent to user programs and requires no special libraries or other utilities. Most programs that operate under UNIX 4.3BSD operate under Mach without modification, after being recompiled.

Mach provides the following features not found in UNIX 4.3BSD:

- Multiple tasks, each with a large, paged virtual memory space
- Multiple threads of execution within each task, with a flexible scheduling facility
- Flexible sharing of memory between tasks
- Efficient and consistent message-based interprocess communication
- Memory-mapped files
- Transparent network extensibility
- A flexible, capability-based approach to security and protection
- Support for multiprocessor scheduling

Mach is sometimes referred to as an object-oriented operating system because it provides most services through user-level programs accessible by a consistent system of message passing. It's important, however, to distinguish between Mach objects and messages and the Objective C objects and messages used in higher-level software kits such as the Application Kit™. Mach objects and messages are distinct from those used in the kits. Kit objects can, however, communicate with the operating system by sending Mach messages to Mach objects or by using the standard UNIX system call interface.

This chapter describes both the Mach kernel and user-level programs that interact with it, but doesn't attempt to redocument standard features of UNIX 4.3BSD. (See the "Suggested Reading" section at the end of this manual for information about available UNIX 4.3BSD documentation.) Individual Mach functions are described in detail in Chapter 4, "Mach Functions," and summarized in the *NeXTSTEP Programming Interface Summary.*

Design Philosophy

Several factors were considered in choosing an operating system for NeXT computers. It was important that the operating system be:

- Multiuser and multitasking
- Network-compatible
- An excellent program-development environment
- Well-represented in the university, research, and business communities
- Extensible and robust
- Capable of providing room for growth and future extensions

Although a standard version of the UNIX operating system would have satisfied many of these criteria, NeXT wanted an operating system offering better performance and a better foundation for future extensions. Mach, with its UNIX 4.3BSD compatibility and improved system design, provided these.

UNIX 4.3BSD compatibility is important because as a multitasking, multiuser operating system, the UNIX environment has gained wide acceptance in many fields, particularly education. Since the creation of the UNIX operating system in 1969, many hours have been spent testing, improving, and extending its features. Currently the UNIX environment is considered one of the best for developing both small and large applications.

However, the success and longevity of the UNIX operating system have exacted their own costs. Many of the features that made the UNIX operating system popular have disappeared in the quest for functionality beyond the scope of the original design. During two decades, the UNIX operating system has grown from a system designed for 16-bit minicomputers without paged memory or networking, to a system that supports multiprocessor mainframes with virtual memory and support for both local and wide-area networks. As a result of these extensions, the UNIX kernel (originally attractive to developers because of its small size, handful of system calls, and ease of modification) has grown to immense proportions.

As new features have been added to the kernel, its size and complexity have grown to the point where its underlying conceptual structure is obscured. Over time, programmers have added multiple routines that perform similar services for different kernel features. The complexity added by each of these extensions ensures that future kernel extensions will be based on an even less sound understanding of what already exists. The result is a system whose complex internal state and interactions make it very difficult to extend, debug, and configure.

Not only has the UNIX kernel grown more complex as new features have been added, so has the interface presented to programmers who would like to make use of these features. For example, current UNIX systems provide an overwhelming variety of interprocess communication (IPC) facilities, including pipes, named pipes, sockets, and message queues. Unfortunately, none of these facilities is general enough to replace the others. As a result, the programmer must understand not only how to use a variety of IPC facilities, but also the tradeoffs involved in choosing one over another.

While retaining UNIX 4.3BSD functionality, Mach departs from current UNIX design and returns to the tenets on which the UNIX operating system was originally built. Foremost among these is the idea that the kernel should be as small as possible, containing only a set of conceptually simple, yet powerful, primitive functions that programmers can use to construct more complex objects.

Mach is designed to put most services provided by the current UNIX kernel into independent user-level programs, with the Mach kernel itself providing only the most basic services:

- Processor scheduling
- Interprocess communication
- Management of virtual memory

These services and others are accessed through a single form of IPC, regardless of whether they're provided by the kernel or by user-level programs. Modularity and a consistent pattern of IPC simplify the interface presented to the programmer. For example, a network expert can implement a new protocol without having to understand or modify other subsystems in the operating system.

Modularity has other advantages as well. Moving functionality to user-level programs makes the kernel smaller and therefore easier to comprehend and debug. Another advantage is the ability to use standard debuggers and other tools to develop new system services rather than having to use special, less powerful tools. Also, configuring the system is simply a matter of choosing which user-level services to initiate, rather than building and linking a new kernel.

The movement of Mach toward providing most operating system features as user-level processes is an evolutionary one. Currently, Mach supports some features within the kernel while others exist at the user level. Although Mach will change as it evolves, its developers are committed to maintaining UNIX 4.3BSD compatibility at each stage of development. If you design your programs to run under UNIX 4.3BSD, they'll run under current and subsequent releases of the Mach operating system. However, if you choose to take advantage of features unique to Mach, future releases of the operating system may require you to modify and recompile some of your programs.

The Mach Kernel

Mach minimizes kernel size by moving most kernel services into user-level processes. The kernel itself contains only the services needed to implement a communication system between various user-level processes. The kernel exports several abstractions to users, including tasks, threads, ports, and messages.

The functionality of the Mach kernel can be divided into the following categories:

- Task and thread creation and management facilities
- Port management facilities

- Basic message functions and support facilities
- Virtual memory management functions
- Scheduling functions

Descriptions of these areas of functionality are provided in the following sections. Messages are described in detail in Chapter 2, "Using Mach Messages."

Mach Tasks and Threads

Mach splits the traditional UNIX notion of a process into two abstractions, the task and the thread:

- A *task* is the environment within which program execution occurs. It's also the basic unit of resource allocation—each task includes a paged virtual address space and port rights that protect access to system resources such as processors, communication capabilities, and virtual memory. The task itself performs no computation; rather, it's a framework for running threads.

- A *thread* is the basic unit of execution. It's a lightweight process executing within a task, and consists solely of the processor state (such as program counter and hardware registers) necessary for independent execution. Each thread executes within the context of a single task, though each task may contain more than one thread. All threads within a task share the virtual memory address space and communication rights of that task.

The task is the basic unit of protection—all threads within a task have access to all that task's capabilities, and aren't protected from each other.

A traditional UNIX process is represented in Mach as a task with a single thread of execution. One major difference between a UNIX process and a Mach task is that creating a new thread in a task is faster and more conservative of system resources than creating a new UNIX process. Creating a new UNIX process involves making a copy of the parent task's address space, but threads share the address space of their task.

Threads are the basic unit of scheduling. On a multiprocessor host, multiple threads from one task may be executing simultaneously within the task's one address space. A thread may be in a *suspended* state (prevented from running), or in a *runnable* state (that is, either currently running or scheduled to run). A nonnegative suspend count is associated with each thread. The suspend count is 0 for runnable threads and positive for suspended threads.

Tasks can be suspended or *resumed* (made runnable) as a whole. A thread can execute only when both it and its task are runnable.

Multiple threads executing within a single task are useful if several program operations need to execute concurrently while accessing the same data. For example, a word processing application could be designed as multiple threads within a single task. The main thread of execution could provide the basic services of the program: formatting text, processing user requests, and so on. Another thread could check the spelling of each word as it's typed in. A third thread could modify the shape of the cursor based on its position within the text window. Since these threads must have access to the same data and should execute concurrently, Mach's design is particularly advantageous.

In addition, threads are well adapted for use with computers that incorporate a multiprocessor architecture. With some multiprocessor machines, individual threads can execute on separate processors, vastly improving overall application performance.

To create and use threads in an application, you should use the C-thread functions. C threads are described later in this chapter; each C-thread function is described in detail in Chapter 4.

Task and Thread Ports

Both tasks and threads are represented by ports. (Ports in Mach are message queues; they're described in the following section.) The task port and the thread port are the arguments used in Mach function calls to identify to the kernel which task or thread is to be affected by the call. The two functions **task_self()** and **thread_self()** return the task and thread ports of the currently executing thread.

Tasks can have access to the task and thread ports of other tasks and threads. For example, a task that creates another task or thread gets access to the new task port or thread port. Also, any thread can pass access to these ports in a message to another thread in the same or a different task.

Having access to a task or thread port enables the possessor to perform Mach function calls on behalf of that task or thread. Access to a task's port indirectly permits access to all threads within that task with the **task_threads()** call; however, access to a thread's port doesn't imply access to its task's port.

The task port and thread port are often called *kernel ports*. In addition to the kernel ports, tasks and threads have a number of special ports associated with them. These are ports that the kernel must know about to communicate with the task or thread in a structured manner.

A task has three ports associated with it, in addition to its kernel port:

- *Notify port*—The port on which the task receives messages from the kernel advising it of changes in port access rights and of the status of messages it has sent. For example, if a thread is unsuccessful in sending a message to another thread's port, its notify port will contain a status message stating that the port has been intentionally destroyed, that the port's task no longer exists, or that there has been a network failure. The task can get this port's value from the function **task_notify()**.

  Note that if a task's notify port is PORT_NULL, no notification messages are generated. This port is set to PORT_NULL at task creation, so a task that wants to receive notifications must explicitly set its notify port with the function **task_set_special_port()**.

- *Exception port*—The port on which the task receives messages from the kernel when an exception occurs. *Exceptions* are synchronous interruptions to the normal flow of program control caused by the program itself. They include illegal memory accesses, protection violations, arithmetic exceptions, and hardware instructions intended to support emulation, debugging, and error detection. Some of these exceptions are handled transparently by the operating system, but some must be reported to the user program. A default exception port is inherited from the parent at task creation time. This port can be changed by the task or any one of its threads in order to take an active role in handling exceptions.

- *Bootstrap port*—The port to which a new task can send a message that will return any other system service ports that the task needs (for example, a port to the Network Name Server). A default bootstrap port is inherited from the parent at task creation. This is the one port that the kernel doesn't actually use; it just makes it available to a new task.

A thread has two ports, in addition to its kernel port:

- *Reply port*—Used in Mach remote procedure calls (remote procedure calls are described in Chapter 2). The **thread_reply()** function returns the reply port of the calling thread.

- *Exception port*—The port to which the kernel sends exceptions occurring in this thread. This port is set to PORT_NULL at thread creation and can be set subsequently with the function **thread_set_exception_port()**. As long as the thread exception port is PORT_NULL, the task exception port is used instead.

Customarily, only threads within a task manipulate that task's state, but this custom isn't enforced by the Mach kernel. A debugger task, for example, can manipulate the state of the task being debugged by getting the task's kernel port and using it in Mach function calls.

Mach Ports and Messages

In Mach, communication among operating system objects is achieved through messages. Mach messaging is implemented by three kernel abstractions:

- *Port*—A protected communication channel (implemented as a finite-length message queue) to which messages may be sent and logically queued until reception. The port is also the basic object reference mechanism in Mach; its use is similar to that of object references in an object-oriented system. That is, operations on objects are requested by sending messages to and from the ports that represent them. When a task is created, a port that represents the task is simultaneously created. When the task is destroyed, its port is also destroyed.

- *Port set*—A group of ports, implemented as a queue combining the message queues of the constituent ports. A thread may use a port set to receive a message sent to any of several ports.

- *Message*—Used to communicate between objects; the message is passed to an object by being sent to the port that represents the object. Each message is a data stream consisting of two parts: a fixed-length header and a variable-length message body composed of zero or more typed data objects. The header contains information about the size of the message, its type, and its destination. The body contains the content (or a pointer to the content) of the message. Messages may be of any size, and may contain in-line data, pointers to data, and capabilities for ports. A single message may transfer up to the entire address space of a task.

Message passing is the primary means of communication both among tasks and between tasks and the kernel. In fact, the only way one object can communicate with another object is by sending a message to that object's port. System services, for example, are invoked by a thread in one task sending a message to another task that provides the desired service. The only functions implemented by system traps are those directly concerned with message communication; all the rest are implemented by messages to the kernel port of a task.

Threads within a single task also use messages and ports to communicate with each other. For example, one thread can suspend or resume the execution of another thread by sending the appropriate message to the thread's port. A thread can also suspend or resume the execution of all threads within another task by sending the appropriate message to the task's port.

The indirection provided by message passing allows objects to be arbitrarily placed in the network without regard to programming details. For example, a thread can suspend another thread by sending a suspend message to the port representing that other thread even if the request is initiated on another node in a network. It's thus possible to run varying system

configurations on different classes of machines while providing a consistent interface to all resources. The actual system running on any particular machine is more a function of its servers than its kernel.

Port Access Rights

Communication between objects is protected by a system of *port access rights*. Access rights to a port consist of the ability to send to or receive from that port. For example, before a task can send a message to a port, it must gain send rights to that port. Before a message can be received, a task must gain receive rights to the port containing the message.

The port access rights operate as follows:

- *Send access* to a port—Implies that a message can be sent to that port. If the port is destroyed during the time a task has send access, the kernel sends a message to that task's notify port indicating that the port has disappeared. For loadable kernel servers, this notification message isn't sent unless the server has requested notification by calling **kern_serv_notify**().

- *Receive access* to a port—Allows a message to be dequeued from that port. Only one task may have receive access for a given port at a time; however, more than one thread within that task may concurrently attempt to receive messages from a given port. When the receive rights to a port are destroyed, that port is destroyed and tasks holding send rights are notified. Receive access implies send rights.

Although multiple tasks may hold send rights to the same port, only one task at a time may hold receive rights to a port.

A thread's right of access is identical to that of the thread's task. Also, when a thread creates a port, send and receive rights are accorded to the task within which the thread is executing. Thus, all threads within the task have equivalent access rights to the new port. Thereafter, any thread within the task can deallocate any or all of these rights, or transfer them to other tasks. The transfer of port rights is accomplished through the Mach messaging system: Access to a port is gained by receiving a message containing a port capability (that is, a capability to either send or receive messages).

Port access rights can be passed in messages. The rights are interpreted by the kernel and transferred from the sender to the kernel upon message transmission and to the receiver upon message reception. Send rights are kept by the original task as well as being transmitted to the receiver task, but receive rights are removed from the original task at the time of the send, and appear in the user task when the receive is done.

During the time between a send and receive, the kernel holds the rights, and any messages sent to the port will be queued waiting for a new task to receive on the port. If the task that was intended to receive the rights dies before it receives them, the rights are handled as though the task had received them before it died.

The type usually used for ports is **port_t**. However, ports can also be referred to as the equivalent types **port_name_t** and **port_all_t**. The **port_name_t** type implies that no port access rights are being transferred; the port is merely being referred to by its name. The **port_all_t** type implies that all rights (both send and receive) for a port are being transferred.

Port Sets

Conceptually, a port set is a bag holding zero or more receive rights. A port set allows a thread to block while waiting for a message sent to any of several ports. A port may be a member of no more than one port set at any time, and a task can have only one port set.

A task's port set right, created by **port_set_allocate()**, allows the task to receive a message from the port set with **msg_receive()** and manipulate the port set with **port_set_add()**, **port_set_remove()**, **port_set_status()**, and **port_set_deallocate()**. Unlike port rights, a port set right can't be passed in messages.

Port set rights usually have the type **port_set_name_t**, which is equivalent to **port_name_t**.

Port Names

Every task has its own port name space, used for port and port set names. For example, one task with receive rights for a port may know the port by the name 13, while another task with send rights for the same port may know it by the name 17. A task has only one name for a port, so if the task with send rights named 17 receives another message carrying send rights for the same port, the arriving rights will also be named 17.

Typically, these names are small integers, but this is implementation dependent. When a task receives a message carrying rights for a new port, the Mach kernel is free to choose any unused name. The **port_rename()** call can be used to change a task's name for a port.

Port Queues

Messages that are sent to a port are held there until removed by a thread. The queue associated with a port is of finite length and may become full. If an attempt is made to send a message to a port that's temporarily full, the sending thread has a choice of three alternatives:

- By default, the sender is suspended until it can successfully transmit the message.
- The sender can have the kernel hold the message for later transmission to the currently full port. If the sender selects this action, it can't transmit further messages to the port (nor can it have the kernel hold additional messages for the port) until the kernel notifies it that the port has received the initial message.
- The attempt to send a message to a full port can simply be reported to the sender as an error.

Extended Communication Functionality

The kernel's message-based communication facility is the building block on which more complicated facilities may be constructed; for example, it's the underlying communication mechanism for the Mach exception-handling facility. Two properties of the Mach communication facility simplify the process of extending the functionality of systems based on it:

- Independence—A port is an independent entity from the tasks that use it to communicate. Port rights can be exchanged in messages, and are tracked by the kernel to maintain protection.
- Network transparency—As described in the following section, user-level network message servers transparently extend the Mach communication facility across a network, allowing messages to be sent between tasks on different computers. The forwarding process is invisible to both the sender and the receiver of the message.

This combination of independence and network transparency enables Mach to support parallel and distributed architectures with no change to the operating system kernel. These properties of the communication facility also simplify the incorporation of new operating system functionality, because user-level programs can easily be added to the existing kernel without the need to modify the underlying kernel base.

Although messaging is similar to UNIX 4.3BSD stream sockets in that it permits reliable, kernel-mediated communication between tasks, messaging has a much more fundamental role within Mach. Whereas UNIX processes obtain system services through a variety of interfaces (for example, the **open()** system call for files, the **socket()** and **bind()** system calls for network connections, and numerous access protocols for user-level services), Mach provides all services through messaging. Because of this consistency of interprocess communication, the Mach operating system can easily be extended to incorporate new features.

As an alternative to messaging, Mach also supports interprocess communication using shared memory. However, if you use shared memory for interprocess communication, you're responsible for synchronizing the transmission and reception of the message. With the Mach messaging system, Mach itself schedules the transmission and reception of messages, thereby ensuring that no message is read before it's been sent in its entirety.

Messaging in a Network Environment

Mach's object-oriented design is well-suited for network operation. Messages may be sent between tasks on different computers just as they're sent between tasks on the same computer. The only difference is the transparent intervention of a new user-level object, the *network server.*

Programs called network servers act as intermediaries for messages sent between tasks on separate computers. Each network server implements *network ports* that represent ports for tasks on remote nodes. A unique *network port identifier* is used to distinguish each network port.

A message addressed to a remote port is first received at the local network port that represents the remote port. The network server, upon receiving the message, translates it into a form compatible with the network protocol and then transmits the message to the counterpart network server on the destination node. The destination server decodes the message, and determines its ultimate destination from the network port identifier in the message. Finally, the destination network server dispatches the message to the local port to which it was addressed.

This network messaging process is transparent to the sender; all routing services are provided by the network server.

Mach Virtual Memory Management

Each Mach task receives a 4-gigabyte virtual address space for its threads to execute in. This address space consists of a series of mappings between ranges of memory addressable to the task and memory objects. Besides accommodating the task and its threads, this space serves as the basis of the Mach messaging system and allows space for memory-mapped files.

A task can modify its address space in several ways. It can:

- Allocate a region of virtual memory (on a page boundary).
- Deallocate a region of virtual memory.
- Set the protection status of a region of virtual memory.
- Specify the inheritance of a region of virtual memory.
- Create and manage a memory object that can then be mapped into the space of another task.

The only restriction imposed by Mach on the nature of the regions that may be specified for virtual memory operations is that they must be aligned on system page boundaries. The size in bytes of a virtual memory page is contained in the **vm_page_size** variable.

Demand Paging

A NeXT computer's memory management hardware is responsible for mapping sections of the virtual memory space into pages of physical memory as needed. The process it uses to decide which virtual pages map to which physical pages is known as *demand paging*.

While a task is executing, only the page of memory containing the addresses referenced by the active thread must reside in physical memory. If the thread references an address not contained in a page of physical memory, the kernel requests the appropriate pager to read in the needed page from storage. Then, a NeXT computer's memory management unit maps the referenced virtual page onto this new physical page of memory.

If there are no further free pages of physical memory available, the Mach kernel makes room by requesting the pager to copy the least recently used page to the paging file on the disk. The kernel then reassigns the newly freed page of memory.

Mach's paged virtual address space makes it possible to run extremely large applications on a NeXT computer. With all but the largest applications, you can continue to allocate memory without concern for exceeding the system's capacity, although to prevent unnecessary performance degradation, you should deallocate memory that's no longer needed.

Inheritance and Protection of Memory

The Mach virtual memory management system also streamlines the creation of a new task (the child) from an existing task (the parent), an operation similar to forking a UNIX process. Traditionally, under the UNIX operating system, creating a new process entails creating a copy of the parent's address space. This is an inefficient operation since often the child task, during its existence, touches only a portion of its copy of the parent's address space. Under Mach, the child task initially shares the parent's address space and copying occurs only when needed, on a page-by-page basis.

A region of an address space represents the memory associated with a continuous range of addresses, marked by a starting address and an ending address. Regions consist of pages that have different protection or inheritance characteristics. The Mach kernel extends each region to include the entire virtual memory pages that contain the starting and ending addresses in the specified range.

Inheritance and protection are attached to a task's address space, not the physical memory contained in that address space. Tasks that share memory may specify different protection or inheritance for their shared regions.

Inheritance

A task may specify that pages of its address space be inherited by child tasks in three ways:

- Copy—Pages marked as copy are logically copied by value, although for efficiency copy-on-write techniques are used. This means the first time the child task attempts to write to shared memory, a protection fault occurs. The kernel responds to this fault by making a copy, for the child task, of the page being written. This is the default mode of inheritance if no mode is specified.

- Shared—Pages specified as shared can be read from and written to by both the parent and child.

- None—Pages marked as none aren't passed to a child. In this case, the child's corresponding address is left unallocated.

Inheritance may be specified globally or on a page-by-page basis when a task is forked. Inheritance may be changed at any time; only at the time of task creation is inheritance information used.

Copy-on-write sharing between unrelated tasks is typically the result of large message transfers. An entire address space may be sent in a single message with no actual data copy operations performed.

Currently the only way two Mach tasks can share the same physical memory is for one of the tasks to inherit shared access to memory from a parent.

Protection

Besides specifying page inheritance attributes, a task may assign protection values to protect the virtual pages of its address space by allowing or preventing access to that memory. Protection values are a combination of read, write, and execute permissions.

By default, when a child task inherits memory from a parent, it gets the same protection on that memory that its parent had.

Like inheritance, protection is specified on a per-page basis. For each group of pages there exist two protection values: the current and the maximum protection. The current protection is used to determine the access rights of an executing thread, and the maximum protection specifies the maximum value that the current protection may take. The maximum value may be lowered but not raised. If the maximum protection is lowered to a level below the current protection, the current protection is also lowered to that level.

For example, a parent task may create a child task and set the maximum protection value for some pages of memory to read-only. Thereafter, the parent task can be assured that the child won't be able to alter the information in those pages.

Interprocess Communication

Mach virtual memory management provides an efficient method of interprocess communication. Messages of any size (up to the limits imposed by the virtual address space) can be transferred between tasks by revising the mapping from the virtual address space of a process to physical address space. This is accomplished by mapping an unused portion of the virtual address space of the receiving process onto the addresses of the sender's message.

The efficiency of this method can be appreciated more fully when compared to the standard UNIX method. Under the UNIX operating system, a message must be physically copied from the address space of the sender into the address space of the kernel. From there, the message is copied into the address space of the receiver.

Memory-Mapped Files

Memory-mapped files are a further benefit of the Mach virtual memory system. Under Mach, all or part of a disk file can be mapped onto a section of virtual memory. A reference to a position within this section is equivalent to a reference to the same position in the physical file. If that portion of the file isn't currently in memory, a page fault occurs, prompting the kernel to request the file system to read the needed section of the file into physical memory. From the point of view of the process, the entire file is in memory at once.

With Mach, the use of memory-mapped files is optional and currently only supports reading files. Mach also supports the standard UNIX **read()**, **lseek()**, and **write()** system calls.

Paging Objects

A *paging object* is a secondary storage object that's mapped into a task's virtual memory. Paging objects are commonly files managed by a file server, but as far as the Mach kernel is concerned, a paging object may be implemented by any port that can handle requests to read and write data.

Physical pages in an address space have paging objects associated with them. These objects identify the backing storage to be used when a page is to be read in as the result of a reference or written to in order to free physical memory.

Virtual Memory Functions

The Mach kernel provides a set of functions to allow a programmer to manipulate the virtual address space of a task. The two most fundamental ones are **vm_allocate()** to get new virtual memory and **vm_deallocate()** to free virtual memory. The programmer also has available the UNIX functions **malloc()**, **calloc()**, and **free()**, which have been reimplemented to use **vm_allocate()** and **vm_deallocate()**.

In addition to memory explicitly allocated using **vm_allocate()**, memory may appear in a task's address space as the result of a **msg_receive()** operation.

The decision to use one allocation method rather than another should be based on several factors. The **vm_allocate()** function always adds new, zero-filled virtual memory in

page-aligned chunks that are multiples of the page size. The **malloc()** function allocates approximately the size asked for (plus a few bytes) out of a preallocated heap. The **calloc()** function is the same as **malloc()** except that it zeros the memory before returning it. Both **malloc()** and **calloc()** are library subroutine calls; **vm_allocate()** is a Mach kernel function, which is somewhat more expensive.

The most obvious basis on which to choose an allocation function is the size of the desired space. One other consideration is the desirability of page-aligned storage. If the memory that's allocated is to be passed *out-of-line* in a message (referred to by a pointer in the message), it's more efficient if it's page-aligned.

Note that it's essential that the correct deallocation function be used. If memory has been allocated with **vm_allocate()**, it must be deallocated with **vm_deallocate()**; if it was allocated with **malloc()** it must be deallocated with **free()**. Memory that's received out-of-line from a message has been allocated by the kernel with **vm_allocate()**.

Program Examples: Virtual Memory

The following three examples demonstrate various aspects of the use of virtual memory functions in C programs.

The first program, **vm_read.c**, demonstrates the use of **vm_allocate()**, **vm_deallocate()**, and another virtual memory function called **vm_read()**. First some memory is allocated and filled with data. The **vm_read()** Mach function is then called, with reading starting at the previously allocated chunk. The contents of the two pieces of memory (that is, the one retrieved by **vm_allocate()** and the one by **vm_read()**) are compared. The **vm_deallocate()** function is then used to get rid of the two chunks of memory.

```
#import <mach/mach.h>
#import <stdio.h>

main(int argc, char *argv[])
{
    char          *data1, *temp;
    char          *data2;
    int            i, min;
    unsigned int  data_cnt;
    kern_return_t rtn;

    if (argc > 1) {
        printf("vm_read takes no switches.  ");
        printf("This program is an example vm_read\n");
        exit(-1);
    }
```

```
if ((rtn = vm_allocate(task_self(), (vm_address_t *)&data1,
        vm_page_size, TRUE)) != KERN_SUCCESS) {
    mach_error("vm_allocate failed", rtn);
    printf("vmread: Exiting.\n");
    exit(-1);
}

temp = data1;
for (i = 0; (i < vm_page_size); i++)
    temp[i] = i;
printf("Filled space allocated with some data.\n");
printf("Doing vm_read....\n");
if ((rtn = vm_read(task_self(), (vm_address_t)data1,
        vm_page_size, (pointer_t *)&data2, &data_cnt))
        != KERN_SUCCESS) {
    mach_error("vm_read failed", rtn);
    printf("vmread: Exiting.\n");
    exit(-1);
}
printf("Successful vm_read.\n");

if (vm_page_size != data_cnt) {
    printf("vmread: Number of bytes read not equal to number");
    printf("available and requested.\n");
}
min = (vm_page_size < data_cnt) ? vm_page_size : data_cnt;

for (i = 0; (i < min); i++) {
    if (data1[i] != data2[i]) {
        printf("vmread: Data not read correctly.\n");
        printf("vmread: Exiting.\n");
        exit(-1);
    }
}
printf("Checked data successfully.\n");

if ((rtn = vm_deallocate(task_self(), (vm_address_t)data1,
        vm_page_size)) != KERN_SUCCESS) {
    mach_error("vm_deallocate failed", rtn);
    printf("vmread: Exiting.\n");
    exit(-1);
}
```

```
    if ((rtn = vm_deallocate(task_self(), (vm_address_t)data2,
            data_cnt)) != KERN_SUCCESS) {
        mach_error("vm_deallocate failed", rtn);
        printf("vmread: Exiting.\n");
        exit(-1);
    }
}
```

The next program, **vm_copy.c**, demonstrates the use of **vm_allocate()**, **vm_deallocate()**, and **vm_copy()**. First, some memory is allocated and filled with data. Then another chunk of memory is allocated, and **vm_copy()** is called to copy the contents of the first chunk to the second. The data in the two spaces is compared to be sure it's the same, checking **vm_copy()**. The **vm_deallocate()** function is then used to get rid of the two chunks of memory.

```
#import <mach/mach.h>
#import <stdio.h>

main(int argc, char *argv[])
{
    int         *data1, *data2, *temp;
    int          i;
    kern_return_t    rtn;

    if (argc > 1) {
        printf("vm_copy takes no switches.    ");
        printf("This program is an example vm_copy\n");
        exit(-1);
    }

    if ((rtn = vm_allocate(task_self(), (vm_address_t *)&data1,
            vm_page_size, TRUE)) != KERN_SUCCESS) {
        mach_error("vm_allocate failed", rtn);
        printf("vm_copy: Exiting.\n");
        exit(-1);
    }

    temp = data1;
    for (i = 0; (i < vm_page_size / sizeof(int)); i++)
        temp[i] = i;
    printf("vm_copy: set data\n");
```

```
    if ((rtn = vm_allocate(task_self(), (vm_address_t *)&data2,
            vm_page_size, TRUE)) != KERN_SUCCESS) {
        mach_error("vm_allocate failed", rtn);
        printf("vm_copy: Exiting.\n");
        exit(-1);
    }

    if ((rtn = vm_copy(task_self(), (vm_address_t)data1, vm_page_size,
            (vm_address_t)data2)) != KERN_SUCCESS) {
        mach_error("vm_copy failed", rtn);
        printf("vm_copy: Exiting.\n");
        exit(-1);
    }
    printf("vm_copy: copied data\n");

    for (i = 0; (i < vm_page_size / sizeof(int)); i++) {
        if (data1[i] != data2[i]) {
            printf("vm_copy: Data not copied correctly.\n");
            printf("vm_copy: Exiting.\n");
            exit(-1);
        }
    }
    printf("vm_copy: Successful vm_copy.\n");

    if ((rtn = vm_deallocate(task_self(), (vm_address_t)data1,
            vm_page_size)) != KERN_SUCCESS) {
        mach_error("vm_deallocate failed", rtn);
        printf("vm_copy: Exiting.\n");
        exit(-1);
    }

    if ((rtn = vm_deallocate(task_self(), (vm_address_t)data2,
            vm_page_size)) != KERN_SUCCESS) {
        mach_error("vm_deallocate failed", rtn);
        printf("vm_copy: Exiting.\n");
        exit(-1);
    }
    printf("vm_copy: Finished successfully!\n");
}
```

The following program, **copy_on_write.c**, demonstrates the use of **vm_inherit()** and copy-on-write memory. A child and parent task share memory, polling this memory to see whose turn it is to proceed.

First, some memory is allocated, and **vm_inherit()** is called on this memory, the variable **lock**. Then more memory is allocated for the copy-on-write test. A fork is executed, and the parent then stores new data in the copy-on-write memory previously allocated, and sets the shared variable signaling to the child that the parent is now waiting. The child, polling the shared variable, sees that the parent is waiting. The child prints the value of the variable **lock** and a value of the copy-on-write memory as the child sees it. The value of **lock** is what the parent set it to be, but the value of the copy-on-write memory is the original value and not what the parent changed it to be. The parent then awakens and prints out the two values once more. The program then ends with the parent signaling the child using the shared variable **lock**.

Typically you wouldn't do this synchronization directly as shown here, but would use C-thread functions (described later in this chapter).

```
#import <mach/mach.h>
#import <stdio.h>

#define NO_ONE_WAIT 0
#define PARENT_WAIT 1
#define CHILD_WAIT 2
#define COPY_ON_WRITE 0
#define PARENT_CHANGED 1
#define CHILD_CHANGED 2
#define MAXDATA 100

main(int argc, char *argv[])
{
    int           pid;
    int          *mem;
    int          *lock;
    kern_return_t ret;

    if (argc > 1) {
        printf("cowtest takes no switches.    ");
        printf("This is an example of copy-on-write \n");
        printf("memory and the use of vm_inherit.\n");
        exit(-1);
    }

    if ((ret = vm_allocate(task_self(), (vm_address_t *)&lock,
            sizeof(int), TRUE)) != KERN_SUCCESS) {
        mach_error("vm_allocate failed:", ret);
        printf("Exiting with error.\n");
        exit(-1);
    }
```

```
if ((ret = vm_inherit(task_self(), (vm_address_t)lock,
        sizeof(int), VM_INHERIT_SHARE)) != KERN_SUCCESS) {
    mach_error("vm_inherit failed:", ret);
    printf("Exiting with error.\n");
    exit(-1);
}

*lock = NO_ONE_WAIT;
if ((ret = vm_allocate(task_self(), (vm_address_t *)&mem,
        sizeof(int) * MAXDATA, TRUE)) != KERN_SUCCESS) {
    mach_error("vm_allocate failed:", ret);
    printf("Exiting with error.\n");
    exit(-1);
}

mem[0] = COPY_ON_WRITE;
printf("value of lock before fork: %d\n", *lock);
pid = fork();

if (pid) {
    printf("PARENT: copied memory = %d\n", mem[0]);
    printf("PARENT: changing to %d\n", PARENT_CHANGED);
    mem[0] = PARENT_CHANGED;
    printf("\n");
    printf("PARENT: lock = %d\n", *lock);
    printf("PARENT: changing lock to %d\n", PARENT_WAIT);
    printf("\n");
    *lock = PARENT_WAIT;
    while (*lock == PARENT_WAIT)
        /* wait for child to change the value */ ;
    printf("PARENT: copied memory = %d\n", mem[0]);
    printf("PARENT: lock = %d\n", *lock);
    printf("PARENT: Finished.\n");
    *lock = PARENT_WAIT;
    exit(-1);
}

while (*lock != PARENT_WAIT)
    /* wait for parent to change lock */ ;
```

```
    printf("CHILD: copied memory = %d\n", mem[0]);
    printf("CHILD: changing to %d\n", CHILD_CHANGED);
    mem[0] = CHILD_CHANGED;
    printf("\n");
    printf("CHILD: lock = %d\n", *lock);
    printf("CHILD: changing lock to %d\n", CHILD_WAIT);
    printf("\n");

    *lock = CHILD_WAIT;
    while (*lock == CHILD_WAIT)
        /* wait for parent to change lock */ ;
    if ((ret = vm_deallocate(task_self(), (vm_address_t)lock,
            sizeof(int))) != KERN_SUCCESS) {
        mach_error("vm_deallocate failed:", ret);
        printf("Exiting.\n");
        exit(-1);
    }

    if ((ret = vm_deallocate(task_self(), (vm_address_t)mem,
            MAXDATA * sizeof(char))) != KERN_SUCCESS) {
        mach_error("vm_deallocate failed:", ret);
        printf("Exiting.\n");
        exit(-1);
    }
    printf("CHILD: Finished.\n");
}
```

Mach Scheduling

Each thread has a scheduling *priority* and *policy*. The priority is a number between 0 and 31 that indicates how likely the thread is to run. The higher the priority, the more likely the thread is to run. For example, a thread with priority 16 is more likely to run than a thread with priority 10. The policy is by default a timesharing policy, which means that whenever the running thread blocks or a certain amount of time passes, the highest-priority runnable thread is executed. Under the timesharing policy, a thread's priority gets lower as it runs (it *ages*), so that not even a high-priority thread can keep a low-priority thread from eventually running.

Priorities

Each thread has three types of priorities associated with it: its base priority, its current priority, and its maximum priority. The base priority is the one the thread starts with; it can be explicitly set using a function such as **cthread_priority()**. The current priority is the one at which the thread is executing; it may be lower than the base priority due to aging or a call to **thread_switch()**. The maximum priority is the highest priority at which the thread can execute. When a thread starts, it inherits its base priority from its parent task and its maximum priority is set to a system-defined maximum.

These priorities can be set at three levels: the thread, the task, and (on multiprocessors) the processor set. At the thread level, you can use **cthread_priority()** or **thread_priority()** to set the base priority and to optionally lower the maximum priority. You can raise or lower just the maximum priority using **cthread_max_priority()** or **thread_max_priority()**. To raise a thread's maximum priority, you must obtain the privileged port of the thread's processor set, which only the superuser can do.

At the task level, you can set the task's base priority using **task_priority()**. The task's base priority is inherited by all threads that it forks; you can also specify that all existing threads in the task get the new base priority.

You can get the priorities of running tasks using **task_info()** and **thread_info()**. Or, from a shell window, you can view the priorities of running tasks using the UNIX command **ps**. The **-l** option of **ps** displays, among other things, the lowest values for maximum priority and current priority that were found in all the threads in the task. The **-m** option displays the current priority of every thread in the task. The following example shows the **ps** displays for Terminal.

```
localhost> ps -axu | grep Terminal
me         1658   2.8  2.4 1.31M  200K p2 S     0:00 grep Terminal
root        174   2.4 11.4 3.84M  936K p1 S     0:41 /NextApps/Terminal.app/T
localhost> ps -l 174
      F  UID   PID  PPID CP PRI BASE VSIZE RSIZE WCHAN STAT TT  TIME COMMAND
      1    0   174   156  0  10   10 4.30M 1.14K     0  S    ?  0:41 /NextAp
localhost> ps -m 174
USER       PID TT  %CPU STAT PRI     SYSTEM      USER  COMMAND
root       174  ?   1.8  S    16   0:15.76   0:19.17  /NextApps/Terminal.app/
                  0.1  S    10   0:06.15   0:00.54
```

Policies

The NeXT Mach operating system has three scheduling policies:

- Timesharing
- Interactive
- Fixed priority

Every thread starts with the timesharing policy, no matter what policy the creator of the thread has. If you want the policy of any thread to be something other than timesharing, you must set that thread's policy using **thread_policy**().

The interactive policy is a variant of timesharing that's designed to be optimized for interactive applications. If you have a non-NeXTSTEP application, such as a terminal-oriented editor, you should set the main thread's policy to interactive using **thread_policy**(). (The Application Kit automatically sets up the first thread in an application to have an interactive policy.) Currently, the interactive policy is exactly the same as timesharing, but in the future performance might be enhanced by, for example, making interactive policy threads have higher priorities than the other threads in the task.

Fixed priority can be a dangerous policy if you aren't familiar with all of its consequences. For this reason, the fixed-priority policy is disabled by default. If you want to use fixed priorities, you must enable them using **processor_set_policy_enable**(). Threads that have the fixed-priority policy have their current priority always equal to their base priority (unless their priority is depressed by **thread_switch**()). A thread with the fixed-priority policy runs until one of the following happens:

- A higher-priority process becomes available to run.
- A per-thread, user-specified amount of time (the *quantum*) passes.
- The thread blocks, waiting for some event or system resource.

Because fixed-priority threads don't lose priority over time, they can prevent lower-priority threads from running. The opposite can happen, too; a low-priority, fixed-priority thread can be kept from running for enough time by higher-priority threads. The first problem can be solved in some cases by the fixed-priority thread calling **thread_switch**() to temporarily depress its priority or hand off the processor to another thread. The fixed-priority policy is often used for real-time problems, such as on-line transaction processing.

Mach C-Thread Functions

Mach provides a set of low-level, language-independent functions for manipulating threads of control. The C-thread functions are higher-level, C language functions in a run-time library that provide an interface to the Mach facilities. The constructs provided in the C-thread functions are:

- Forking and joining of threads
- Protection of critical regions with mutual exclusion (mutex) variables
- Condition variables for synchronization of threads

Another way of protecting critical regions and synchronizing threads is to use the Mach Kit's locking classes. The Mach Kit provides an object-oriented interface to a few limited areas of Mach functionality. See the *NeXTSTEP General Reference* for information on the Mach Kit.

If you intend to build multithreaded applications, you should use the C-thread functions rather than the Mach kernel functions. The C-thread functions are a natural and efficient set of functions for multithreaded applications, whereas the Mach thread functions are designed to provide the low-level mechanisms that packages such as the C-thread functions can be built with. An on-line example of a multithreaded NeXTSTEP application is under the directory **/NextDeveloper/Examples/AppKit/SortingInAction**.

Using External Functions and Methods

Many of the functions and methods provided by NeXTSTEP weren't designed with multithreaded applications in mind. As a result, they might not work correctly when called simultaneously by two or more of your application's threads. (A function or method that can safely be called by more than one thread at once is *thread-safe*.) In general, unless you know that a function or method is thread-safe, you should assume that it isn't.

The following are thread-safe:

- Distributed Objects, which is described in the *General Reference*
- Mach functions (except for **mach_error()**)
- UNIX system calls (you should use **cthread_errno()** instead of **errno**)
- NeXTSTEP exception handling (for example, **NX_RAISE()**)
- The **malloc()** function and its related functions, though their thread safety can be disabled by calling **malloc_singlethreaded()** (which in general should not be done)

The Objective C runtime system is not thread-safe by default. To make it thread-safe, use the function **objc_setMultithreaded()**.

The following are *not* thread-safe:

- The Application Kit (messages to kit objects should be sent only from the main thread)
- DPS client routines
- The Window Server (drawing should be done only from the main thread)
- Standard I/O functions, such as **printf()**
- Most of the functions in the libc library

Using Shared Variables

All global and static variables are shared among all threads: If one thread modifies such a variable, all other threads will observe the new value. In addition, a variable reachable from a pointer is shared among all threads that can dereference that pointer. This includes objects pointed to by shared variables of pointer type, as well as arguments passed by reference in **cthread_fork()**. You should be careful to declare all shared variables as **volatile**, or the optimizer might remove references to them.

When pointers are shared, some care is required to avoid problems with dangling references. You must ensure that the lifetime of the object pointed to is long enough to allow the other threads to dereference the pointer. Since there's no bound on the relative execution speed of threads, the simplest solution is to share pointers to global or heap-allocated objects only. If a pointer to a local variable is shared, the function that variable is defined in must remain active until it can be guaranteed that the pointer will no longer be dereferenced by other threads. The synchronization functions can be used to ensure this.

Unless a library has been designed to work in the presence of reentrancy, you should assume that the library makes unprotected use of shared data. You must protect against this through the use of a mutex that's locked before every library call (or sequence of library calls) and unlocked afterward. For example, you should lock a mutex before calling **printf()** and unlock the mutex afterward.

Synchronization of Variables

This section describes mutual exclusion and synchronization functions, which are used to constrain the possible interleavings of the execution streams of threads. These functions manipulate *mutex* and *condition* variables, which are defined as follows:

```
typedef struct mutex {...} *mutex_t;

typedef struct condition {...} *condition_t;
```

Mutually exclusive access to mutable data is necessary to prevent corruption of data. As a simple example, consider concurrent attempts to update a simple counter. If two threads fetch the current value into a (thread-local) register, increment, and write the value back in some order, the counter will only be incremented once, losing one thread's operation. A mutex solves this problem by making the fetch-increment-deposit action atomic. Before fetching a counter, a thread locks the associated mutex, and after depositing a new value the thread unlocks the mutex:

```
mutex_lock(m);
count += 1;
mutex_unlock(m);
```

If any other thread tries to use the counter in the meantime, it will block when it tries to lock the mutex. If more than one thread tries to lock the mutex at the same time, only one will succeed; the rest will block.

Condition variables are used when one thread wants to wait until another thread has finished doing something. Every condition variable should be protected by a mutex. Conceptually, the condition is a boolean function of the shared data that the mutex protects. Commonly, a thread locks the mutex and inspects the shared data. If it doesn't like what it finds, it waits, using a condition variable:

```
mutex_lock(mutex_t m);
. . .
while ( /* condition isn't true */ )
    condition_wait(condition_t c, mutex_t m);
. . .
mutex_unlock(mutex_t m);
```

The call to **condition_wait()** temporarily unlocks the mutex to give other threads a chance to get in and modify the shared data. Eventually, one of them should signal the condition (which wakes up the blocked thread) before it unlocks the mutex:

```
mutex_lock(mutex_t m);
. . .       /* modify shared data */
condition_signal(condition_t c);
mutex_unlock(mutex_t m);
```

At that point, the original thread will regain its lock and can look at the shared data to see if things have improved. It can't assume that it will like what it sees, because some other thread may have slipped in and altered the data after the condition was signaled.

You must take special care with data structures that are dynamically allocated and deallocated. In particular, if the mutex that's controlling access to a dynamically allocated record is part of the record, make sure that no thread is waiting for the mutex before freeing the record.

Attempting to lock a mutex that one already holds is another common error. The offending thread will block waiting for itself. This can happen when a thread is traversing a complicated data structure, locking as it goes, and reaches the same data by different paths. Another instance of this is when a thread is locking elements in an array, say to swap them, and it doesn't check for the special case that the elements are the same.

You must be careful to avoid deadlock, a condition in which one or more threads are permanently blocked waiting for each other. The above scenarios are a special case of deadlock. The easiest way to avoid deadlock with mutexes is to impose a total ordering on the mutexes, and then ensure that threads only lock mutexes in increasing order.

You must decide what kind of granularity to use in protecting shared data with mutexes. The two extremes are to have one mutex protecting all shared memory, or to have one mutex for every byte of shared memory. Finer granularity normally increases the possible parallelism because less data is locked at any one time. However, it also increases the overhead lost to locking and unlocking mutexes and increases the possibility of deadlock.

Program Example: C Threads

This section demonstrates the use of the C-thread functions in writing a multithreaded program. The program is an example of how to structure a program with a single master thread that spawns a number of concurrent slaves. The master thread waits until all the slaves have finished and then exits.

Once created, a slave thread simply loops calling a function that makes the processor available to other threads. After this loop is finished, the slave thread informs the master that it's done, and then dies. In a more useful version of this program, each slave process would do something while looping.

```
#import <stdio.h>
#import <mach/cthreads.h>

volatile int count;     /* number of slave threads active */
mutex_t      lock;      /* mutual exclusion for count */
mutex_t      print;     /* mutual exclusion for printfs */
condition_t  done;      /* signaled each time a slave finishes */

void init()
{
    /* Allocate mutex variables "lock" and "print". */
    lock = mutex_alloc();
    print = mutex_alloc();

    /* Allocate condition variable "done". */
    done = condition_alloc();

    count = 0;
}

/*
 * Each slave just loops, yielding the processor on each
 * iteration.  When it's finished, it decrements the global
 * count and signals that it's done.
 */
void slave(int n)
{
    int i;

    for (i = 0; i < 100; i += 1)
        cthread_yield();
```

```
    /*
     * If any thread wants to access the count variable, it
     * first locks the mutex.  When the mutex is locked, any
     * other thread wanting the count variable must wait until
     * the mutex is unlocked.
     */
    mutex_lock(lock);
    count -= 1;
    mutex_lock(print);
    printf("Slave %d finished.\n", n);
    mutex_unlock(print);
    /* Signal that this slave has finished. */
    condition_signal(done);
    mutex_unlock(lock);
}

/*
 * The master spawns a given number of slaves and then waits
 * for them all to finish.
 */
void master(int nslaves)
{
    int i;

    for (i = 1; i <= nslaves; i++) {
        mutex_lock(lock);
        /* Increment count with the creation of each slave thread.  */
        count += 1;
        /* Fork a slave and detach it.  */
        cthread_detach(cthread_fork((cthread_fn_t)slave, (any_t)i));
        mutex_unlock(lock);
    }

    mutex_lock(lock);
    /*
     * Master thread loops waiting on the condition done.  Each
     * time the master thread is signaled by a condition_signal
     * call, it tests the count for a value of zero.
     */
    while (count != 0)
        condition_wait(done, lock);
    mutex_unlock(lock);

    mutex_lock(print);
    printf("All %d slaves have finished.\n", nslaves);
    mutex_unlock(print);
}
```

```
main()
{
    init();
    master(15);  /* Create master thread and 15 slaves. */
}
```

Mach Exception Handling

Exceptions are synchronous interruptions to the normal flow of program control caused by the occurrence of unusual conditions during program execution. Raising an exception causes the operating system to manage recovery from the unusual condition.

Exceptions include:

- Illegal accesses (bus errors, segmentation and protection violations)
- Arithmetic errors (overflow, underflow, divide by zero)
- Hardware instructions intended to support facilities such as emulation, debugging, and error detection

Note: Software interrupts and other actions caused by asynchronous external events aren't considered to be exceptions.

Although many exceptions, such as page faults, can be handled by the operating system and dismissed transparently to the user, the remaining exceptions are exported to the user by the operating system's exception-handling facility (for example, by invoking a handler or producing a core dump).

Four major classes of applications use exceptions:

- Debugging—Debuggers rely on exceptions generated by hardware trace and breakpoint facilities. Other exceptions that indicate errors must be reported to the debugger; the presence of the debugger indicates the user's interest in any anomalous program behavior.
- Core dumps—In the absence of a debugger, a fatal exception can cause the execution state of a program to be saved in a file for later examination.

- Error handling—Certain applications sometimes handle their own exceptions (particularly arithmetic). For example, an error handler could substitute 0 for the result of a floating underflow and continue execution. Error handlers are often required by high-level languages.

- Emulation—Generally, computers generate exceptions upon encountering operation codes that can't be executed by the hardware. Emulators can be built to execute the desired operation in software. Such emulators serve to extend the instruction set of the underlying machine by performing instructions that aren't present in the hardware.

The following sections contrast the UNIX approach to error handling with the general model upon which the Mach exception-handling facility is built, and then present specific information about the Mach exception-handling facility.

The UNIX Approach to Exception Handling

Designers of operating systems have approached exceptions in a variety of ways. The drawbacks of most approaches include limited functionality (often the result of designing only for debuggers) and lack of extensibility to a multithreaded environment.

The UNIX operating system generalizes exception handling to the signal facility, which handles all interruptions to normal program flow. The varying requirements of different types of interruptions (such as exceptions, timer expiration, or a control character from the terminal) entail semantics that vary from signal to signal; the default action can be nothing, stop, continue from stop, or terminate (with or without a core dump). The user can change these defaults or specify a handler to be invoked by a signal. The interface to these handlers includes a partial machine context, but registers outside this context aren't accessible.

Debugging support in UNIX is centralized in the **ptrace()** system call: It performs all data transfer and process control needed by debuggers, and interacts with the signal facility to make signals visible to debuggers (including signals that would otherwise invoke error handlers or emulators). The occurrence of a signal in a debugged process causes that process to stop in a peculiar manner and notify the debugger that something has happened. This notification is implemented by special treatment of debugged processes in the **wait()** system call; this call usually detects terminated processes, but also detects stopped processes that are being debugged. One consequence of these features and their implementation is that debuggers are restricted to debugging processes that are the immediate children of the debugger.

Two major problems with the UNIX signal facility are:

- Executing the signal handler in the same context as the exception makes many registers inaccessible. These registers are often the very registers that an arithmetic error handler needs to modify (for example, by substituting 0 for a floating underflow).

- The entire concept of signals is predicated on single-threaded applications. Adapting signals to multithreaded applications is difficult and complicates the interface to them. At least half a dozen major changes to the UNIX signal implementation in the Mach kernel have been required for this reason.

The typical use of signal handlers is to detect and respond to external events; for this they're adequate, but as an exception-handling facility, they leave much to be desired.

A Model for Generalized Exception Handling

The Mach exception-handling facility is based on a model whose generality is sufficient to describe virtually all uses of exceptions, including those made by the four classes of applications discussed earlier.

The Mach exception-handling model divides applications that use exceptions into two major classes:

- Error handlers—Perform recovery actions in response to an exception and resume execution of the thread involved. This class includes both error handlers and emulators. Error handlers typically execute in the same address space as that thread for efficiency reasons (access to state).

- Debuggers—Examine the state of an entire application to investigate why an exception occurred or why the program is misbehaving. This class includes both interactive debuggers and the servers that produce core dumps; the latter can be viewed as front ends to debuggers that examine core dumps. Debuggers usually execute in address spaces distinct from the application for protection reasons.

This chapter uses the terms *error handler* and *debugger* to refer to these two classes (for example, a core dumper is a debugger). The term *handler* is used to refer to any application that uses exceptions.

The Mach exception-handling model is derived by examining the requirements common to error handlers and debuggers. Specifically, the occurrence of an exception requires suspension of the thread involved and notification of a handler. The handler receives the notification and performs some computation (for example, an error handler fixes the error, a debugger decides what to do next), after which the thread is either resumed or terminated.

The model presented in this section covers all uses of exceptions. The occurrence of an exception invokes a four-step process involving the thread that caused the exception (victim) and the entity that handles the exception (handler, which may be the operating system):

1. Victim does a *raise*, causing notification of the occurrence of an exception.

2. Victim does a *wait*, synchronizing with completion of exception handling.

3. Handler does a *catch*, receiving notification. This notification usually identifies the exception and the victim, although some of this identification may be implicit in where and how the notification is received.

4. Handler takes either of two possible actions: *clear* the exception (causing the victim to return from the *wait*), or *terminate* the victim thread.

The primitives appearing in bold in this model constitute the high-level model interface to exceptions and can be viewed as operating on *exception objects*. The handler will usually perform other functions between the *catch* step and the *clear* or *terminate* step; these functions are part of the handler application itself, rather than part of the exception model.

Exception Handling in Mach

The Mach exception-handling facility was designed as a general implementation of the exception-handling model described above. The major design goals for this new facility were:

- Single facility with consistent semantics for all exceptions
- Clean and simple interface
- Full support for debuggers and error handlers
- No duplication of functionality within kernel
- Support for user-defined exceptions

A consequence of these goals is a rejection of the notion of a handler executing in the same context as the exception it's handling. There is no clean and straightforward way to make a thread's context available to the thread itself; this results in a single thread having multiple contexts (a currently executing context and one or more saved exception contexts). In turn this causes serious naming and functionality problems for operations that access or manipulate thread contexts. Because Mach supports multiple threads within the same task, it's sufficient to stop the thread that caused the exception and execute the handler as another thread in the same task.

The Mach exception-handling facility implements the exception-handling model with Mach kernel functions to avoid duplicating kernel functionality. Because the handler never executes in the context of the victim thread, the *raise*, *wait*, *notify*, and *clear* primitives constitute a remote procedure call (RPC). They're therefore implemented using a message-based RPC provided by the Mach communication facility. The remaining *terminate* primitive is exactly the **thread_terminate()** or **task_terminate()** function; no special action is required to terminate the thread or task instead of completing the RPC.

The exception RPC consists of two messages: an initial message to invoke the RPC, and a reply message to complete the RPC. The initial message contains the following items:

- Send and reply ports for the RPC
- The identities of the thread that caused the exception and of the corresponding task
- A machine-independent exception class (see the section "Exception Classification")
- Two machine-dependent fields that further identify the exception

If the RPC is completed, the reply message contains the two RPC ports and a return code from the handler that handled the exception (success in almost all cases). MiG-generated stub routines perform the generation and decoding of the messages; this allows users to avoid dealing directly with the contents of the messages. (MiG is described in Chapter 2.)

An exception RPC corresponds to our exception model as follows:

- *raise*—Send initial message.
- *wait*—Wait for and receive reply message.
- *catch*—Receive initial message.
- *clear*—Send reply message.

Exception Ports

The two messages that constitute the RPC are sent to and received from ports corresponding to the handler (initial message) and victim (reply message). The handler's port is registered as the exception port for either the victim's task or thread; the kernel consults this registration when an exception occurs. The reply port is specified in the initial message; for hardware exceptions, the kernel allocates the reply port and caches it for reuse on a per-thread basis. Mach kernel functions are available to register a port as an exception port for a task or thread, and to return the port currently registered; these functions for implementing debuggers and error handlers are described in the section "Program Example: Exception Handling."

Registering exception ports for both tasks and threads effects a separation of concerns between error handlers and debuggers. Error handlers are supported by the thread exception ports because error handlers usually affect only the victim thread; different

threads within a task can have different error handlers. The registered exception port for a thread defaults to the null port at thread creation; this defaults the initial error handler to no handler. Debuggers are supported by the task exception ports because debuggers operate on the application level; this includes at least all the threads in the victim's task, so at most one debugger is associated with a single task. The registered exception port for a task is inherited from the parent task at task creation; this supports debuggers that handle trees of tasks (such as a multitasking parallel program) and inheritance of core-dump servers.

The presence of both task and thread exception ports creates a potential conflict because both are applicable to any exception. This is resolved by examining the differences between error handlers and debuggers. Error handlers use exceptions to implement portions of an application; an error handler is an integral part of the application that generates its exceptions. Exceptions handled by an error handler may be unusual, but they don't indicate anomalous or erroneous behavior. In contrast, debuggers use exceptions to investigate anomalous or erroneous application behavior; as a result debuggers have little interest in exceptions successfully handled by error handlers. This implies that exceptions should invoke error handlers in preference to debuggers; this preference is implemented by having thread exception ports take precedence over task exception ports in determining where to direct the RPC invoked by an exception. If neither an error handler nor a debugger can successfully handle an exception, the task is terminated.

User Extensibility

The Mach exception-handling facility permits you to define and handle your own exceptions in addition to those defined by the system.

The software class of exceptions (see the section "Exception Classification") contains a range of codes reserved for user-defined exceptions; this allows the handling of these exceptions to be integrated into the handling of system-defined exceptions. The same ports are used in both cases, and the interface to handlers is identical.

An advantage of this approach is that user-defined exceptions can immediately be recognized as such, even by debuggers that can't decode the machine-dependent fields that identify the exact exception.

Generation of user-defined exceptions is facilitated by a MiG stub routine that implements the exception RPC (in turn this routine is generated automatically from an interface description of the exception RPC). User code that detects an exception simply obtains the appropriate exception port from the kernel and calls this stub routine; the stub routine handles the RPC and returns a return code from the handler. Alternatively, you may use the MiG exception interface with your own exceptions and exception ports; this approach may be advantageous for applications that handle only user-defined exceptions.

Implementing Error Handlers

Error handlers are supported by thread exception ports and invoked by remote procedure calls on those ports. An error handler is associated with a thread by registering a port on which the error handler receives exception RPCs as the exception port of the thread. This registration causes all exceptions occurring in the thread to invoke RPCs to the error handler's port. Since most error handlers can't handle all possible exceptions that could occur, they must check each exception and forward it to the corresponding task exception port if it can't be handled. This forwarding can be performed by obtaining the exception port for the task specified in the initial message and sending the initial message there. Alternatively, the error handler can return a failure code in the reply message; this causes the sender of the initial message to reinitiate the RPC using the task exception port.

Implementation of error handlers requires additional functionality beyond completing the RPC. This functionality is supported by separate Mach kernel functions that can also be used by other applications. The most common actions and corresponding functions are:

- Read/write register state: **thread_get_state()**, **thread_set_state()**
- Read/write memory state: Access memory directly within task; otherwise **vm_read()**, **vm_write()**
- Terminate thread: **thread_terminate()**
- Resume thread: Send reply message to complete RPC (**msg_send()**)

Some applications may require that error handlers execute in the context of (that is, on the stack of) the thread that caused the exception (such as emulation of UNIX signal handlers). Although this appears to conflict with the principle of never executing an error handler in the context of the victim thread, it can be implemented by using a system-invoked error handler to set up the application's handler. Specifically, the error handler invoked by the exception RPC modifies the victim thread so that the application's handler is executed when the thread is resumed. Unwinding the stack when the application's error handler finishes is the responsibility of the application developer.

Implementing Debuggers

Debuggers are supported by the task exception ports; exceptions invoke debuggers with remote procedure calls on those ports. A debugger is associated with a task by registering a port on which the debugger receives exception RPCs as the task's exception port. An exception RPC stops only the victim thread pending RPC completion; other threads in the task continue running. This has two consequences:

- If the debugger wants to stop the entire task, a **task_suspend()** must be performed. A straightforward way to accomplish this is to do it inside the exception RPC and then complete the RPC; the victim thread can't resume execution upon RPC completion because its task has been suspended.

- Multiple exceptions from a multithreaded task may be outstanding for the debugger on a single debugger invocation. If the debugger doesn't handle these pending exceptions for the task, some may appear to occur at impossible times (such as a breakpoint occurring after the user has removed it).

The Mach exception-handling facility is one small component of the kernel that can be used by debuggers. The various actions required to support debuggers are implemented using general-purpose functions that also support other applications. Some of the more important debugger actions and corresponding kernel functions are:

- Detect event: **msg_receive()**. System components that generate or detect external events (such as interrupt characters on a terminal) signal the events by sending messages.

- Read and write application memory (includes setting breakpoints): **vm_read()**, **vm_write()**.

- Read and write application registers (includes setting single-step mode if available): **thread_get_state()**, **thread_set_state()**.

- Continue application: Task and thread control functions.

- End debugging session: **task_terminate()**.

Exceptions that invoke error handlers using thread exception ports aren't visible to debuggers. A debugger that wants to detect error handler invocation can insert one or more breakpoints in the error handler itself; exceptions caused by these breakpoints will be reported to the debugger.

Debugger Attachment

The independence property of the Mach kernel described previously allows Mach to support debugger attachment and detachment without change to the kernel itself. Traditional UNIX systems require that the debugged process be the child of the debugger; this makes it impossible to debug a process that wasn't started by the debugger. Subsequent developers have expended considerable effort to implement an **attach** primitive that allows a debugger to attach to a previously started process and debug it; this allows analysis of failures that may not be repeatable. Similarly these systems allow a debugger to detach from a running process and exit without affecting the process. No design change is required to support this functionality; the debugger need only obtain the port representing the task to be debugged, and may then use all of the functions previously discussed to debug that task. A debugger can detach from a task by resetting the task's exception port to its former value; there is no other connection between the debugger and task being debugged.

Parallel and Distributed Debugging

The design of the exception-handling facility also supports parallel and distributed debugging without change. There are several cases to be considered based on the structure of the debugger and the application being debugged. In all of these cases the debugger itself may be a parallel or distributed application consisting of multiple tasks and threads.

For parallel applications composed of multiple threads within a single task, a debugger need only register its exception RPC port as that task's exception port. Multiple concurrent exceptions result in multiple RPC invocations being queued to that port; each invocation identifies the thread involved. The Mach communication facility allows the debugger to accept all of these RPCs before responding to any of them, and to respond to them in any order. (Of course the debugger must keep track of the RPCs and make sure they're all responded to when continuing the application.) A straightforward implementation is to suspend the task in response to the first RPC, and then complete all pending exception RPCs recording the threads and exceptions involved. The exceptions can then be reported to the user all at once.

For parallel applications composed of multiple tasks within a single machine, only minor changes to the above debugger logic are required. The debugger must now register its exception RPC port as the task exception port for each task, and may choose to identify components of the parallel application by tasks instead of threads. Suspending or resuming the entire application now requires an operation on each task. If the application dynamically creates tasks, an additional interface to report these new tasks to the debugger may be required so that the new tasks can be suspended and resumed by the debugger.

Network transparency allows the components of a debugger and the debugged application to be spread throughout a network; all required operations extend transparently across the network. This supports a number of possible debugging scenarios:

- The application and the debugger are on separate hosts.
- The application being debugged is distributed over the network. The debugger doesn't require modifications beyond those needed to deal with applications composed of multiple tasks.
- The debugger itself can be distributed over the network.

The last scenario is useful for implementing fast exception response in a debugger for applications that run in parallel on several distributed hosts; if the exception RPC stays within the host, suspending of all application components on that host can be done faster.

GDB Enhancements

The Mach exception-handling facility and other Mach kernel functions have been used to enhance GDB (the GNU source-level debugger) for debugging multithreaded tasks. This enhanced version of GDB operates at the task level (that is, any exception causes GDB to suspend the entire task). A notion of the current thread has been added; this thread is used by any thread-specific command that doesn't specify a thread. New commands are provided to list the threads in the task, change the current thread, and examine or control individual threads. Thread-specific breakpoints are supported by logic that transparently continues the application from the breakpoint until the desired thread hits it. Implementation of attachment to running tasks, as described earlier in the section "Debugger Attachment," is in progress, as are changes to deal with multiple concurrent breakpoints.

The existence of multiple threads within a debugged task complicates GDB's execution control logic. In addition to the **task_suspend()** required upon exception detection, resuming from a breakpoint becomes somewhat intricate. Standard GDB removes the breakpoint, single-steps the process, puts back the breakpoint and continues. The enhanced version must ensure that only the thread at the breakpoint executes while performing the single step; this requires switching from task suspension to suspension of all of the threads except one and then back again before resuming the application.

The Mach exception-handling facility is an important implementation base for the enhancements to GDB. Identification of the victim thread in the initial message makes it possible to handle multiple concurrent exceptions; all the UNIX functions (for example, **ptrace()**) are restricted to one current signal per task, and hence preclude handling of

multiple concurrent exceptions. Additionally, the independence of the debugger from the debugged application makes it possible to implement debugger attachment without kernel modifications; the UNIX operating system requires extensive kernel modifications to achieve similar functionality.

Exception Classification

The Mach exception-handling facility employs a new hardware-independent classification of exceptions. This is in contrast to previous systems (such as UNIX), whose exception classifications are closely wedded to the hardware they were originally developed on. This new classification divides all exceptions into six classes based on the causes and uses of the exceptions; further hardware-specific and software-specific distinctions can be made within these classes as needed. The six classes are:

- Bad access—A user access to memory failed for some reason and the operating system was unable to recover (such as invalid memory or protection violation).
- Bad instruction—A user executed an illegitimate instruction (such as an undefined instruction, reserved operand, or privileged instruction).
- Arithmetic—A user arithmetic instruction failed for an arithmetic reason (such as overflow, underflow, or divide by zero).
- Emulation—A user executed an instruction requiring software emulation.
- Software—A broad class including all exceptions intended to support software. These fall into three subclasses:

Hardware	Hardware instructions to support error detection (such as trap on overflow or trap on subscript out of range).
Operating system	Exceptions detected by operating system during system call execution (such as no receiver on pipe). These are for operating system emulation (such as UNIX emulation). Mach doesn't use exceptions for system call errors.
User	Exceptions defined and caused by user software for its own purposes.

- Debugger—Hardware exceptions to support debuggers (such as breakpoint instruction and trace trap).

In cases of potential confusion (for example, is it a bad instruction or an instruction requiring emulation?) the correct classification is always clear from the intended uses of the instruction as determined by the hardware and system designers.

Two machine-dependent fields are used to identify the precise exception within a class for flexibility in encoding exception numbers. Two fields are needed for emulation instructions containing a single argument (one for the instruction, one for the argument), but we have also found them useful for constructing machine-dependent exception classifications (for example, using one field to hold the trap number or vector, and the other to distinguish this trap from the others that use this number or vector). Cases in which two fields don't suffice require a separate interface to extract the additional machine-dependent status.

Kernel Interface

This section lists functions that relate directly to the exception-handling facility. The following Mach functions let you raise exceptions, handle them, and get or set exception ports. See Chapter 4 for descriptions of each of these functions and macros.

- **exception_raise()**
- **exc_server()**
- **mach_NeXT_exception()**
- **mach_NeXT_exception_string()**
- **task_set_exception_port()**
- **task_get_exception_port()**
- **thread_set_exception_port()**
- **thread_get_exception_port()**

Another important function is one you implement yourself: **catch_exception_raise()**. If you implement this function, it must have the following syntax:

kern_return_t **catch_exception_raise**(port_t *exception_port*, port_t *thread*, port_t *task*,
int *exception*, int *code*, int *subcode*)

Program Example: Exception Handling

The following example shows how to raise and handle user-defined exceptions. The program sets up a new exception port, sets up a thread to listen to this port, and then raises an exception by calling **exception_raise()**. The thread that's listening to the exception port receives the exception message and passes it to **exc_server()**, which calls the user-implemented function **catch_exception_raise()**.

This program's implementation of **catch_exception_raise()** determines whether it understands the exception. If so, it handles the exception by displaying a message. If not, this implementation of **catch_exception_raise()** sets a global variable that indicates that its calling thread should forward the exception to the old exception port. This program doesn't know which exception handler is listening to the old exception port; it could be the default UNIX exception handler, GDB, or any other exception handler.

```
/*
 * raise.c: This program shows how to raise user-specified exceptions.
 * If you use GDB, you can't set any breakpoints or step through any
 * code between the call to task_set_exception_port and the return
 * from exception_raise().  (You can never use GDB to debug exception
 * handling code, since GDB stops the program by generating an
 * EXC_BREAKPOINT exception.)
 */
#import <mach/mach.h>
#import <mach/exception.h>
#import <mach/cthreads.h>
#import <mach/mig_errors.h>

typedef struct {
    port_t old_exc_port;
    port_t clear_port;
    port_t exc_port;
} ports_t;

volatile boolean_t pass_on = FALSE;
mutex_t            printing;

/* Listen on the exception port. */
any_t exc_thread(ports_t *port_p)
{
    kern_return_t   r;
    char           *msg_data[2][64];
    msg_header_t   *imsg = (msg_header_t *)msg_data[0],
                   *omsg = (msg_header_t *)msg_data[1];
```

```
    /* Wait for exceptions. */
    while (1) {
        imsg->msg_size = 64;
        imsg->msg_local_port = port_p->exc_port;
        r = msg_receive(imsg, MSG_OPTION_NONE, 0);

        if (r==RCV_SUCCESS) {
            /* Give the message to the Mach exception server. */
            if (exc_server(imsg, omsg)) {
                /* Send the reply message that exc_serv gave us. */
                r = msg_send(omsg, MSG_OPTION_NONE, 0);
                if (r != SEND_SUCCESS) {
                    mach_error("exc_thread msg_send", r);
                    exit(1);
                }
            }
            else { /* exc_server refused to handle imsg. */
                mutex_lock(printing);
                printf("exc_server didn't like the message\n");
                mutex_unlock(printing);
                exit(2);
            }
        }
        else { /* msg_receive() returned an error. */
                mach_error("exc_thread msg_receive", r);
                exit(3);
        }

        /* Pass the message to old exception handler, if necessary. */
        if (pass_on == TRUE) {
            imsg->msg_remote_port = port_p->old_exc_port;
            imsg->msg_local_port = port_p->clear_port;
            r = msg_send(imsg, MSG_OPTION_NONE, 0);
            if (r != SEND_SUCCESS) {
                mach_error("msg_send to old_exc_port", r);
                exit(4);
            }
        }
    }
}
```

```
/*
 * catch_exception_raise() is called by exc_server().  The only
 * exception it can handle is EXC_SOFTWARE.
 */
kern_return_t catch_exception_raise(port_t exception_port,
    port_t thread, port_t task, int exception, int code, int subcode)
{
    if ((exception == EXC_SOFTWARE) && (code == 0x20000)) {
        pass_on = FALSE;
        /* Handle the exception so that the program can continue. */
        mutex_lock(printing);
        printf("Handling the exception\n");
        mutex_unlock(printing);
        return KERN_SUCCESS;
    }
    else { /* Pass the exception on to the old port. */
        pass_on = TRUE;
        mutex_lock(printing);
        mach_NeXT_exception("Forwarding exception", exception,
            code, subcode);
        mutex_unlock(printing);
        return KERN_FAILURE;  /* Couldn't handle this exception. */
    }
}

main()
{
    int             i;
    kern_return_t   r;
    ports_t         ports;

    printing = mutex_alloc();

    /* Save the old exception port for this task. */
    r = task_get_exception_port(task_self(), &(ports.old_exc_port));
    if (r != KERN_SUCCESS) {
        mach_error("task_get_exception_port", r);
        exit(1);
    }
```

```
    /* Create a new exception port for this task. */
    r = port_allocate(task_self(), &(ports.exc_port));
    if (r != KERN_SUCCESS) {
        mach_error("port_allocate 0", r);
        exit(1);
    }
    r = task_set_exception_port(task_self(), (ports.exc_port));
    if (r != KERN_SUCCESS) {
        mach_error("task_set_exception_port", r);
        exit(1);
    }

    /* Fork the thread that listens to the exception port. */
    cthread_detach(cthread_fork((cthread_fn_t)exc_thread,
        (any_t)&ports));
    /* Raise the exception. */
    ports.clear_port = thread_reply();
#ifdef NOT_OUR_EXCEPTION
    /* By default, EXC_BAD_ACCESS causes a core dump. */
    r = exception_raise(ports.exc_port, ports.clear_port,
        thread_self(), task_self(), EXC_BAD_ACCESS, 0, 0);
#else
    r = exception_raise(ports.exc_port, ports.clear_port,
        thread_self(), task_self(), EXC_SOFTWARE, 0x20000, 0);
#endif

    if (r != KERN_SUCCESS)
        mach_error("catch_exception_raise didn't handle exception",
            r);
    else {
        mutex_lock(printing);
        printf("Successfully called exception_raise\n");
        mutex_unlock(printing);
    }

    sleep(5);  /* Exiting too soon can disturb other exception
                * handlers. */
}
```

2 *Using Mach Messages*

This chapter describes how to use Mach messages for interprocess communication (IPC). Programs can either send and receive Mach messages directly, or they can use remote procedure calls (RPCs) generated by MiG (Mach Interface Generator). MiG-generated RPCs appear to be simple function calls but actually involve messages. Many kernel functions, such as **host_info()**, are really RPCs.

This chapter first describes the structure of all messages. It then discusses how to set up messages for direct sending. Finally, it discusses how to use MiG to build a *Mach server*—a program that provides services to clients by using remote procedure calls. This chapter assumes that you understand the concepts of ports, port sets, and messages, which are described in Chapter 1, "Mach Concepts."

You should usually use MiG to generate messages. MiG-generated code is easier for clients to use, and using MiG is a good way to define an interface that's separate from the implementation. However, you might want to build messages by hand if the messages are very simple or if you want fine control over communication details.

Note: Tasks can also communicate with each other using Distributed Objects. See the *NeXTSTEP General Reference* for information on Distributed Objects.

Message Structure

A message consists of a fixed header often followed by the message body. The body consists of alternating type descriptors and data items. Here's a typical message structure:

```
typedef struct {
    msg_header_t Head;
    msg_type_t   aType;
    int          a;
    msg_type_t   bType;
    int          b;
} Request;
```

Message Header

The C type definition for the message header is as follows (from the header file **mach/message.h**):

```
typedef struct {
        unsigned int msg_unused : 24,
                     msg_simple : 8;
        unsigned int msg_size;
        int          msg_type;
        port_t       msg_local_port;
        port_t       msg_remote_port;
        int          msg_id;
} msg_header_t;
```

The **msg_simple** field indicates whether the message is *simple* or *nonsimple*; the message is simple if its body contains neither ports nor out-of-line data (pointers).

The **msg_size** field specifies the size of the message to be sent, or the maximum size of the message that can be received. When a message is received, Mach sets **msg_size** to the size of the received message. The size includes the header and in-line data and is given in bytes.

The **msg_type** field specifies the general type of the message. For hand-built messages, it's MSG_TYPE_NORMAL; MiG-generated servers use the type MSG_TYPE_RPC. Other values for the **msg_type** field are defined in the header files **mach/message.h** and **mach/msg_type.h**.

The **msg_local_port** and **msg_remote_port** fields name the ports on which a message is to be received or sent. Before a message is sent, **msg_local_port** must be set to the port to which a reply, if any, should be sent; **msg_remote_port** must specify the port to which the

message is being sent. Before a message is received, **msg_local_port** must be set to the port or port set to receive on. When a message is received, Mach sets **msg_local_port** to the port the message is received on, and **msg_remote_port** to the port any reply should be sent to (the sender's **msg_local_port**).

The **msg_id** field can be used to identify the meaning of the message to the intended recipient. For example, a program that can send two kinds of messages should set the **msg_id** field to indicate to the receiver which kind of message is being sent. MiG automatically generates values for the **msg_id** field.

Message Body

The body of a message consists of an array of type descriptors and data. Each type descriptor contains the following structure:

```
typedef struct  {
    unsigned int
        msg_type_name : MSG_TYPE_BYTE, /* Type of data */
        msg_type_size : 8,             /* Number of bits per item */
        msg_type_number : 12,          /* Number of items */
        msg_type_inline : 1,           /* If true, data follows; else
                                          a ptr to data follows */
        msg_type_longform : 1,         /* Name, size, number follow */
        msg_type_deallocate : 1,       /* Deallocate port rights or
                                          memory */
        msg_type_unused : 1;
} msg_type_t;
```

The **msg_type_name** field describes the basic type of data comprising this object. The system-defined data types include:

- Ports, including combinations of send and receive rights.
- Port and port set names. This is the same language data type as port rights, but the message only carries a task's name for a port and doesn't cause any transferral of rights.
- Simple data types, such as integers, characters, and floating-point values.

The **msg_type_size** field indicates the size in bits of the basic object named in the **msg_type_name** field.

The **msg_type_number** field indicates the number of items of the basic data type present after the type descriptor.

The **msg_type_inline** field indicates that the actual data is included after the type descriptor; otherwise, the word following the descriptor is a pointer to the data to be sent.

The **msg_type_longform** field indicates that the name, size, and number fields were too long to fit into the **msg_type_t** structure. These fields instead follow the **msg_type_t** structure, and the type descriptor consists of a **msg_type_long_t**:

```
typedef struct {
    msg_type_t  msg_type_header;
    short       msg_type_long_name;
    short       msg_type_long_size;
    int         msg_type_long_number;
} msg_type_long_t;
```

When **msg_type_deallocate** is nonzero, it indicates that Mach should deallocate this data item from the sender's address space after the message is queued. You can deallocate only port rights or out-of-line data.

A data item, an array of data items, or a pointer to data follows each type descriptor.

Creating Messages by Hand

This section shows how to create messages to be sent using **msg_send()** or **msg_rpc()**. You don't usually have to set up messages by hand. For example, although Mach servers call **msg_send()**, almost all the message fields are already set up in MiG-generated code. However, this section might be useful if you want to send messages without using MiG, or if you want to read through MiG-generated code.

Setting Up a Simple Message

As described earlier, a message is *simple* if its body doesn't contain any ports or out-of-line data (pointers). The **msg_remote_port** field must contain the port the message is to be sent to. The **msg_local_port** field should be set to the port a reply message (if any) is expected on.

The following example shows the creation of a simple message. Because every item in the body of the message is of the same type (**int**), only one type descriptor is necessary, even though the items are in two different fields.

```
#define BEGIN_MSG 0 /* Constants to identify the different messages */
#define END_MSG 1
#define REPLY_MSG 2

#define MAXDATA 3

struct simp_msg_struct {
    msg_header_t  h;                /* message header */
    msg_type_t    t;                /* type descriptor */
    int           inline_data1;     /* start of data array */
    int           inline_data2[2];
};
struct simp_msg_struct  msg_xmt;
port_t                  comm_port, reply_port;

/* Fill in the message header. */
msg_xmt.h.msg_simple = TRUE;
msg_xmt.h.msg_size = sizeof(struct simp_msg_struct);
msg_xmt.h.msg_type = MSG_TYPE_NORMAL;
msg_xmt.h.msg_local_port = reply_port;
msg_xmt.h.msg_remote_port = comm_port;
msg_xmt.h.msg_id = BEGIN_MSG;

/* Fill in the type descriptor. */
msg_xmt.t.msg_type_name = MSG_TYPE_INTEGER_32;
msg_xmt.t.msg_type_size = 32;
msg_xmt.t.msg_type_number = MAXDATA;
msg_xmt.t.msg_type_inline = TRUE;
msg_xmt.t.msg_type_longform = FALSE;
msg_xmt.t.msg_type_deallocate = FALSE;

/* Fill in the array of data items. */
msg_xmt.inline_data1 = value1;
msg_xmt.inline_data2[1] = value2;
msg_xmt.inline_data2[2] = value3;
```

Setting Up a Nonsimple Message

A message is *nonsimple* if its body contains ports or out-of-line data. The most common reason for sending data out-of-line is that the data block is very large or of variable size.

In-line data is copied by the sender into the message structure and then often copied out of the message by the receiver. *Out-of-line data*, however, is mapped by the kernel from the address space of the sender to the address space of the receiver. No actual copying of out-of-line data is done unless one of the two tasks subsequently modifies the data.

This example shows how to construct a message containing out-of-line data:

```
#define BEGIN_MSG 0 /* Constants to identify the different messages */
#define END_MSG 1
#define REPLY_MSG 2

#define MAXDATA 3

struct ool_msg_struct {
    msg_header_t  h;                /* message header */
    msg_type_t    t;                /* type descriptor */
    int          *out_of_line_data; /* address of data */
};
struct ool_msg_struct   msg_xmt;
port_t                  comm_port, reply_port;

/* Fill in the message header. */
msg_xmt.h.msg_simple = FALSE;
msg_xmt.h.msg_size = sizeof(struct ool_msg_struct);
msg_xmt.h.msg_type = MSG_TYPE_NORMAL;
msg_xmt.h.msg_local_port = reply_port;
msg_xmt.h.msg_remote_port = comm_port;
msg_xmt.h.msg_id = BEGIN_MSG;

/* Fill in the type descriptor. */
msg_xmt.t.msg_type_name = MSG_TYPE_INTEGER_32;
msg_xmt.t.msg_type_size = 32;
msg_xmt.t.msg_type_number = MAXDATA;
msg_xmt.t.msg_type_inline = FALSE;
msg_xmt.t.msg_type_longform = FALSE;
msg_xmt.t.msg_type_deallocate = FALSE;

/* Fill in the out-of-line data. */
msg_xmt.out_of_line_data = (int *)&mydata;
```

The fields that change values from those in the simple message example are **msg_simple**, **msg_type_inline**, and possibly **msg_type_deallocate**. The **msg_type_name**, **msg_type_size**, and **msg_type_number** fields remain the same as before, so that Mach can determine how much memory to map.

The **msg_remote_port** field must contain the port the message is to be sent to. The **msg_local_port** field should be set to the port where a reply message (if any) is expected.

Setting Up a Reply Message

Once a message has been received, a reply message may have to be sent to the sender of the received message. In the following example, the reply message, **msg_xmt**, is simply a **msg_header_t** since no data is required. The **msg_remote_port** field, which designates where to send the message, must be set to the remote port of the previously received message (which Mach set to the previous sender's **msg_local_port** field). The **msg_local_port** field of the outgoing message is set to PORT_NULL because no reply to this message is expected.

```
#define BEGIN_MSG 0 /* Constants to identify the different messages */
#define END_MSG 1
#define REPLY_MSG 2

struct simp_msg_struct {            /* format of received message */
    msg_header_t  h;                /* message header */
    msg_type_t    t;                /* type descriptor */
    int           inline_data1;     /* start of data array */
    int           inline_data2[2];
};
msg_header_t             msg_xmt;
struct simp_msg_struct  *msg_rcv;

msg_xmt.h.msg_remote_port = msg_rcv->h.msg_remote_port;
msg_xmt.h.msg_local_port = PORT_NULL;  /* no reply expected */
msg_xmt.h.msg_id = REPLY_MSG;
msg_xmt.h.msg_size = sizeof(msg_header_t);
msg_xmt.h.msg_type = MSG_TYPE_NORMAL;
msg_xmt.h.msg_simple = TRUE;
```

Mach Interface Generator

The Mach Interface Generator (known as MiG) is a program that generates remote procedure call (RPC) code for communication between a client and a server process. The operations of sending a message and receiving a reply are represented as a single remote procedure call.

For example, if a program makes a call to **host_info()**, it actually calls a library routine that sends a message to the Mach kernel and then waits to receive a reply message. After the Mach kernel sends a reply message containing the information, the library routine takes the data out of the reply message and returns it to the program in parameters to the **host_info()** call. However, the program sees none of this complexity—it merely makes the following function call:

```
ret = host_info(host_self(), HOST_SCHED_INFO,
                (host_info_t)&sched_info, &sched_count);
```

A Mach server executes as a separate task and communicates with its clients by sending Mach messages. As you can see from the previous sections in this chapter, Mach messages are fairly complex. The MiG program is designed to automatically generate procedures in C to pack and send, or receive and unpack the messages used to communicate between processes.

Because of the complexity of sending and decoding messages, Mach remote procedure calls are an order of magnitude slower than real function calls, even if the server is on the local machine. Calls to servers on remote machines take longer. However, Mach RPC has the advantages of the separation of interface and implementation, and of network transparency.

Using MiG, you can create RPC interfaces for sending messages between tasks on the local machine, or between tasks on separate machines in a network. In the network environment, MiG both encodes messages to be transmitted and decodes them upon arrival at the destination node, taking into account dissimilarities in machine architecture.

MiG is especially useful if you're faced with a mixed network environment. Without MiG, you're responsible for providing routines to translate messages between two machines with different data representations. Using MiG, you need only specify the calling arguments of the procedure and the procedure's return variables. The low-level routines required to translate messages between these machines are then generated automatically.

MiG is flexible enough to describe most data structures that might be sent as messages between processes. MiG supports the data types boolean, character, signed and unsigned integers, integer subranges, strings, reals, and communication port types. MiG also supports the limited creation of new data types through the use of enumerations, fixed-size and variable-size arrays, records, pointers to these types, and unions.

Creating Mach Servers with MiG

To create a Mach server, you must provide a specification file defining parameters of both the message-passing interface and the procedure-call interface. MiG then generates three files from the specification file:

- User interface file (*xxx***User.c**, where *xxx* is the subsystem name)—Should be compiled and linked into the client program. It implements and exports the procedures and functions for sending and receiving the appropriate messages to and from the server.

- User header file (*xxx*.**h**)—Defines the functions to be called by a client of the server. It's included in the user interface file (*xxx***User.c**) and defines the types and routines needed at compilation time.

- Server interface file (*xxx***Server.c**)—Should be compiled and linked into the server process. It extracts the input parameters from an IPC message, and calls a server procedure to perform the operation. When the server procedure or function returns, the Server interface also gathers the output parameters and formats a reply message.

Besides the specification file, you must write at least two functions for the Mach server. One is the main routine of the server, which registers the server and then goes into a loop that receives a message, calls the MiG-generated code to process the request, and sends a reply message. You must also write one function for each remote procedure call, so that the MiG-generated server code can call the appropriate function for each request.

In addition, you should provide a library routine that clients can use to look up your server. For example, the kernel-server loader has a routine called **kern_loader_look_up()** that clients call to obtain the kernel-server loader's port. This port must be specified as the first argument in every RPC to the kernel-server loader.

You can register your server with either the Network Name Server or the Bootstrap Server, depending on whether you want your server to be available to other machines on a network. The Bootstrap Server allows only processes that are on the local machine (or a subset of local processes) to get your server's port. For example, the sound driver registers its port with the Bootstrap Server so that only processes descended from the local machine's Login Window can control sound. The Network Name Server allows tasks on remote machines to get the server's port. See Chapter 4, "Mach Functions," for more information on Network Name Server and Bootstrap Server functions.

Client's View

This section describes how clients use servers, so that you can better create and document your own server.

Before a client can make remote procedure calls to the server, it must find the server's port. If the server doesn't provide a library function to do this lookup, then the client must call either **netname_look_up()** or **bootstrap_look_up()** and supply the name of the server.

When a client makes a remote procedure call, it appears to be a simple function call. The return type depends on whether the RPC is defined in the server's MiG specification file to be a routine, procedure, or function (as described later in this chapter).

The most convenient interfaces are to routines, which return a value of type **kern_return_t**. The returned value is either KERN_SUCCESS or a MiG, Mach, or server-specific error code. MiG and Mach error codes can be interpreted by **mach_error()** and **mach_error_string()**.

Procedure and function RPCs are less convenient than routines because they don't directly return error codes. Instead, the client must provide an error-handling routine named either **MsgError()** or whatever name the server developer specified in the server's MiG specification file. The error-handling routine must be defined as follows:

void *error_proc*(kern_return_t *error_code*)

Common Error Codes

The most common system error that an RPC returns to a client is an invalid port. This can mean several things:

- The request port (usually the first parameter in the RPC) is an invalid port, or the client doesn't have send rights for it.
- The reply port is invalid or lacks receive rights. (This problem can't occur unless the client provides the reply port; usually the system provides it.)
- Another port that the client is passing in the message is invalid.
- A port that's being passed back to the client is invalid.

Another system error a client might receive is a timeout. This can happen only if a timeout is specified in an argument or in the server's specification file, and usually doesn't happen unless the server is on a different machine from the client.

MiG errors, which are defined in the header file **mach/mig_errors.h**, usually occur only if the client is using a different version of the interface than the server.

Out-of-Line Data

When making specific interface calls, the client should be aware if any out-of-line data is being returned to it. If so, it might want to deallocate the space with a call to **vm_deallocate()**.

Compiling the Client

The client must be compiled and linked with the *xxx***User.c** and *xxx***.h** files that MiG produced from the server's specification file. The client should also include or be linked with any files that are necessary to communicate with the server (such as the file containing the routine that looks up the server). For example, clients of the kernel-server loader must be linked against the kernload library, which supplies all non-RPC kernel-server loader functions.

Programming Example

This example shows the implementation of a simple server that adds two or three integers and returns the answer. The files used to produce this server and a sample client program are under the **MiG** directory of **/NextLibrary/Documentation/NextDev/Examples**.

The user-written files required for the server are the following:

- MiG specification file (**Server/add.defs**)
- Type definition file, which is included by both the server and the client (**Library/add_types.h**)
- Implementation file, which contains the server's main loop and the function that does the addition (**Server/add_server_main.c**)

Once the server has been generated, any client programs need to have the following files:

- MiG-generated user interface file, in a form that can be compiled or linked into the client program (**Library/addUser.o**)
- MiG-generated user header file (**Library/add.h**)

- Type definition file (**Library/add_types.h**)
- One or more files containing the main parts of the client program (**Client/add.c**)

In the following example, a simple MiG specification file called **add.defs** is shown. It declares the remote routines **add2nums**() and **add3nums**(), which take as arguments the request port (the default first argument to every MiG operation), two or three integers, and a pointer to another integer. Because all types mentioned in **add.defs** are already defined in the included header file **mach/std_types.defs**, it isn't necessary to define any types directly in **add.defs**.

```
/* add.defs:  MiG definition file for add server */

subsystem add 0;

/* Get standard definitions of int and port_t. */
#include <mach/std_types.defs>

routine add2nums(server: port_t; a:int; b:int; out c:int);
routine add3nums(server: port_t; a:int; b:int; c:int; out d:int);
```

The header file **mach/std_types.defs** defines **int** and **port_t** as the following:

```
type int = MSG_TYPE_INTEGER_32;
type port_t = MSG_TYPE_PORT;
```

The header file **add_types.h** contains definitions needed by both the client and the server:

```
/* add_types.h:  Definitions for add server */
#import <mach/mach.h>

#define ADD_SERVER_NAME "Addition-Server"

extern port_t add_look_up(void);
```

The code that does the work for the server is in the file **add_server_main.c**. It contains a main loop and the functions that perform the addition. The main loop dynamically allocates the memory needed for incoming and outgoing messages, using the **addMaxRequestSize** and **addMaxReplySize** constants generated by MiG.

```
/* add_server_main.c:  Main loop and implementation of add server */

#import <mach/mach.h>
#import <mach/message.h>
#import <servers/netname.h>
#import <mach/mach_error.h>
#import <ansi/stdlib.h>
#import "../Library/add_types.h"
#import "addServer.h"

void         server_loop(port_t port);
/* defined by MiG: */
boolean_t    add_server(msg_header_t *in, msg_header_t *out);

void main(int argc, char *argv[])
{
    port_t        server_port;
    kern_return_t r;

    /* Register with the Network Name Server. */
    r = port_allocate(task_self(), &server_port);
    if (r != KERN_SUCCESS) {
        mach_error("port_allocate failed", r);
        exit(1);
    }
    r = netname_check_in(name_server_port, ADD_SERVER_NAME,
        PORT_NULL, server_port);
    if (r != KERN_SUCCESS) {
        mach_error("netname_check_in failed", r);
        exit(1);
    }

    /* Enter our main loop. */
    server_loop(server_port);
}

void server_loop(port_t port)
{
    kern_return_t ret;
    msg_header_t *msg = (msg_header_t *)malloc(addMaxRequestSize);
    msg_header_t *reply = (msg_header_t *)malloc(addMaxReplySize);
```

```
    while (TRUE)
    {
        /* Receive a request from a client. */
        msg->msg_local_port = port;
        msg->msg_size = addMaxRequestSize;
        ret = msg_receive(msg, MSG_OPTION_NONE, 0);
        if (ret != RCV_SUCCESS) /* ignore errors */;

        /* Feed the request into the server. */
        (void)add_server(msg, reply);

        /* Send a reply to the client. */
        reply->msg_local_port = port;
        ret = msg_send(reply, MSG_OPTION_NONE, 0);
        if (ret != SEND_SUCCESS) /* ignore errors */;
    }
}

/*
 * This function is called by add_server, which was created by MiG.
 * It is NOT directly called by any client process.
 */
kern_return_t add2nums(port_t server, int n1, int n2, int *n3)
{
    *n3 = n1+n2;
    return KERN_SUCCESS;
}

/*
 * This function is called by add_server, which was created by MiG.
 * It is NOT directly called by any client process.
 */
kern_return_t add3nums(port_t server, int n1, int n2, int n3, int *n4)
{
    *n4 = n1+n2+n3;
    return KERN_SUCCESS;
}
```

In general, your message receive loop should return a reply for every message it receives unless the reply message returned from the MiG-generated server has MIG_NO_REPLY in its **RetCode** field. MIG_NO_REPLY is used only when the received message was part of an RPC that never expects a return message (a **simpleprocedure** or **simplefunction**, both of which are defined later in this chapter). For example:

```
(void)add_server(msg, reply);
ret_code = reply.RetCode;

if (ret_code == MIG_NO_REPLY)
    ret_code = KERN_SUCCESS;
else
    ret_code = msg_send(reply, MSG_OPTION_NONE, 0);
```

Finally, a typical client process, such as **Client/add.c**, makes the RPC as follows:

```
. . .
#import "../Library/add_types.h"
#import "../Library/add.h"
int           n1, n2, n3, result;
kern_return_t ret;
port_t        server;
. . .
/* Find the server. */
server = add_look_up();
if (server == PORT_NULL)
{
    fprintf(stderr, "Couldn't find the add server.\n");
    exit(2);
}

/* Send a message to the server. */
if (argc == 3) {  /* 2 numbers to add */
    ret = add2nums(server, n1, n2, &result);
    if (ret != KERN_SUCCESS)
        printf("Call to add2nums failed.\n");
    else
        printf("According to the server, %d + %d = %d.\n", n1, n2,
            result);
} else {          /* 3 numbers to add */
    ret = add3nums(server, n1, n2, n3, &result);
    if (ret != KERN_SUCCESS)
        printf("Call to add3nums failed.\n");
    else
        printf("According to the server, %d + %d + %d= %d.\n", n1,
            n2, n3, result);
}
```

Note that although the RPC looks like it directly calls **add2nums()** and **add3nums()** in the server, it really doesn't. The client instead sends a message that's received in **server_loop()**, which calls **add_server()**. The **add_server()** function calls **add2nums()** or **add3nums()** and passes the result back to the client in a message.

Making a function such as **add2nums()** an RPC gives the advantages of network independence, interface independence, and automatic type checking, at the expense of some complexity in the server.

MiG Specification File

You must first write a MiG specification file to specify the details of the procedure arguments and the messages to be used. A MiG specification file contains the following components, some of which may be omitted:

- Subsystem identification
- Type declarations
- Import declarations
- Operation descriptions
- Options declarations

The subsystem identification should appear first for clarity. Types must be declared before they're used. Code is generated for the operations and import declarations in the order in which they appear in the specification files. Options affect the operations that follow them.

See the earlier section, "Programming Example," for a complete subsystem definition.

Subsystem Identification

The subsystem identification statement has the following form:

subsystem *sys message_base_id* ;

The *sys* is the name of the subsystem. It's used as the prefix for all generated file names. The user file name will be *sys***User.c**, the user header file will be *sys***.h**, and the server file will be *sys***Server.c**.

The *message_base_id* is a decimal integer that's used as the IPC message ID of the first operation in the specification file. Operations are numbered sequentially beginning with this base. The MiG-generated server function checks the message ID of an incoming message to make sure that it's no less than *message_base_id* and no greater than *message_base_id* + *num_messages* −1, where *num_messages* is the number of messages understood by the server.

Several servers can use just one message receive loop as long as they have different subsystem numbers (and they have few enough messages so that message IDs don't overlap). The message receive loop should call each MiG-generated server function in turn until one of them returns true (indicating the message ID is in the range understood by that server.) Once a MiG-generated server function has returned true or all the servers have returned false, the receive-serve-send loop should send a reply (unless the reply message returned by the server function has MIG_NO_REPLY in its **RetCode** field).

Example:

```
subsystem random 500;
```

Type Declarations

Simple Types

A simple type declaration has the following form:

type *user_type_name* = *type_desc* [*translation_info*]

where a *type_desc* is either a previously defined *user_type_name* or an *ipc_type_desc,* which has one of the following forms:

ipc_type_name
(*ipc_type_name* [, *size* [, **dealloc**]])

The *user_type_name* is the name of a C type that will be used for some parameters of the calls exported by the user interface file. The *ipc_type_desc* of simple types are enclosed in parentheses and consist of an IPC type name, decimal integer, or integer expression that's the number of bits in the IPC type and, optionally, the **dealloc** keyword.

The standard system-defined IPC type names are:

MSG_TYPE_BOOLEAN
MSG_TYPE_BIT
MSG_TYPE_BYTE
MSG_TYPE_CHAR
MSG_TYPE_INTEGER_8
MSG_TYPE_INTEGER_16
MSG_TYPE_INTEGER_32
MSG_TYPE_REAL
MSG_TYPE_STRING
MSG_TYPE_PORT
MSG_TYPE_PORT_ALL
MSG_TYPE_UNSTRUCTURED

The current set of these type names is contained in the header file **mach/message.h**, which defines all the message-related types needed by a user of the Mach kernel. The programmer may define additional types. If the *ipc_type_name* is a system-defined one other than MSG_TYPE_STRING, MSG_TYPE_UNSTRUCTURED, or MSG_TYPE_REAL, *size* (the bit length) need not be specified and the parentheses can be omitted.

The **dealloc** keyword controls the treatment of ports and pointers after the messages they're associated with have been sent. The **dealloc** keyword causes the deallocation bit in the IPC message to be set on; otherwise, it's off. If **dealloc** is used with a port, the port is deallocated after the message is sent. If **dealloc** is used with a pointer, the memory that the pointer references will be deallocated after the message has been sent. An error results if **dealloc** is used with any argument other than a port or a pointer.

Some examples of simple type declarations are:

```
type int = MSG_TYPE_INTEGER_32;
type my_string = (MSG_TYPE_STRING,8*80);
type kern_return_t = int;
type disposable_port = (MSG_TYPE_PORT_ALL,32,dealloc);
```

The MiG-generated code assumes that the C types **my_string**, **kern_return_t**, and **disposable_port** are defined in a compatible way by a programmer-provided header file. The basic C and Mach types are defined in the header file **mach/std_types.defs**.

MiG assumes that any variable of type MSG_TYPE_STRING is declared as a C **char *** or **char** *array*[*n*]. Thus it generates code for a parameter passed by reference and uses **strncpy()** for assignment statements.

Optional *translation_info* information describing procedures for translating or deallocating values of the type may appear after the type definition information:

- Translation functions, **intran** and **outtran**, allow the type as seen by the user process and the server process to be different.
- Destructor functions allow the server code to automatically deallocate input types after they have been used.

For example:

```
type task_t = (MSG_TYPE_PORT,32)
intran:     i_task_t PortToTask(task_t)
outtran:    task_t TaskToPort(i_task_t)
destructor: DeallocT(i_task_t)
;
```

Note: Because *translation_info* is part of the type declaration, the semicolon (;) doesn't appear until after the end of *translation_info*.

In this example, **task_t**, which is the type seen by the user code, is defined as a port in the message. The type seen by the server code is **i_task_t**, which is a data structure used by the server to store information about each task it's serving. The **intran** function **PortToTask()** translates values of type **task_t** to **i_task_t** on receipt by the server process. The **outtran** function **TaskToPort()** translates values of type **i_task_t** to type **task_t** before return. The destructor function **DeallocT()** is called on the translated input parameter, **i_task_type**, after the return from the server procedure and can be used to deallocate any or all parts of the internal variable. The destructor function won't be called if the parameter is also an **out** parameter (as described later in this chapter, in the section "Operation Descriptions"); this is because the correct time to deallocate an **out** parameter is after the reply message has been sent, which MiG doesn't do. A destructor function can also be used independently of the translation routines. For example, if a large out-of-line data segment is passed to the server, it could use a destructor function to deallocate the memory after the data was used.

Although calls to these functions are generated automatically by MiG, the function definitions must be hand-coded and imported using:

```
i_task_t PortToTask(task_t x)
task_t TaskToPort(i_task_t y)
void DeallocT(i_task_t y)
```

Structured Types

Three kinds of structured types are recognized: arrays, structures, and pointers. Definitions of arrays and structures have the following syntax:

array [*size*] **of** *comp_type_desc*
array [* : *maxsize*] **of** *comp_type_desc*
struct [*size*] **of** *comp_type_desc*

where *comp_type_desc* may be a simple *type_desc* or may be an **array** or **struct** type, and *size* may be a decimal integer constant or expression. The second array form specifies that a variable-length array is to be passed in-line in the message. In this form *maxsize* is the maximum length of the item. Currently, only one variable-length array may be passed per message. For variable-length arrays an additional count parameter is generated to specify how much of the array is actually being used.

If a type is declared as an **array**, the C type must also be an array, since the MiG RPC code will treat the user type as an array (that is, MiG will assume that the user type is passed by reference and it will generate special code for array assignments). A variable declared as a **struct** is assumed to be passed by value and treated as a C structure in assignment statements. There is no way to specify the fields of a C structure to MiG. The *size* and *type_desc* are used only to give the size of the structure. The following example shows how to declare a C structure as a **struct**.

```
/* declaration in MiG .defs file */
type short = MSG_TYPE_INTEGER_16;
type port_t = MSG_TYPE_PORT;
type lock_struct = struct [9] of short;
routine fl_message(server_port: port_t; inout arg: lock_struct);

/* declaration in C code */
typedef struct {
    short l_type;
    short l_whence;
    long l_start;
    long l_len;
    short l_pid;
    long l_hostid;
} lock_struct;
```

Pointer Types

In the definition of pointer types, the symbol ^ precedes a simple, array, or structure definition.

^ *comp_type_desc*
^ **array** [*size*] **of** *comp_type_desc*
^ **struct** [*size*] **of** *com_type_desc*

The *size* may be left blank or be *. In either case, the array or structure is of variable size, and a parameter is defined immediately following the array parameter to contain its size. Data types declared as pointers are sent out-of-line in the message. Since sending out-of-line data is considerably more expensive than sending in-line data, pointer types should be used only for large or variable amounts of data. A call that returns an out-of-line item allocates the necessary space in the user's virtual memory. It's up to the user to call **vm_deallocate**() on this memory when finished with the data.

Some examples of complex types are:

```
type procids = array [10] of int;
type procidinfo = struct [5*10] of (MSG_TYPE_INTEGER_32);
type vardata = array [ * : 1024 ] of int;
type array_by_value = struct [1] of array [20] of (MSG_TYPE_CHAR);
type page_ptr = ^ array [4096] of (MSG_TYPE_INTEGER_32);
type var_array = ^ array [] of int;
```

Import Declarations

If any of the *user_type_name*s or *server_type_name*s are other than the standard C types (such as **int** and **char**), C type specification files must be imported into the user interface and server interface files so that they'll compile. The import declarations specify files that are imported into the modules generated by MiG.

An import declaration has one of the following forms:

import *file_name*;
uimport *file_name*;
simport *file_name*;

where *file_name* has the same form as file name specifications in **#include** statements (that is, <*file_name*> or "*file_name*").

For example:

```
import "my_defs.h";
import "/usr/include/mach/cthreads.h";
import <mach/cthreads.h>;
```

Files included with **import** are included in both the user-side and server-side code. Those included with **uimport** are included in just the user side. Those included with **simport** are included in just the server side.

Operation Descriptions

Any of five standard operations may be specified by using the following keywords:

function
routine
procedure
simpleprocedure
simpleroutine

One other keyword, **skip**, may be used in place of a standard operation.

Functions and routines have a return value; procedures don't. Routines are functions whose result is of type **kern_return_t**. This result indicates whether the requested operation was successfully completed. If a routine returns a value other than KERN_SUCCESS, the reply message won't include any of the reply parameters except the error code. Neither procedures nor functions return indications of errors directly; instead they call a hand-coded error function in the client. The name of the error function is **MsgError()**, by default; you can specify another name using the **error** declaration in the MiG specification file.

Simple procedures and simple routines send a message to the server but don't expect a reply. The return value of a simple routine is the value returned by the function **msg_send()**. Simple routines or simple procedures are used when asynchronous communication with a server is desired. The rest of the operations wait for a reply before returning to the caller.

The syntax of the **procedure**, **simpleprocedure**, **simpleroutine**, and **routine** statements are identical. The syntax of **function** is also the same except for the type name of the value of the function. The general syntax of an operation definition for everything except **function** has the following form:

operation_type operation_name (*parameter_list*) ;

For **function** the form is:

function *operation_name* (*parameter_list*) : *function_value_type* ;

The *parameter_list* is a list of parameter names and types separated by a semicolon. The form of each parameter is:

[*specification*] *var_name* : *type_description* [, **dealloc**]

If not omitted, *specification* must be one of the following:

in
out
inout
requestport
replyport
waittime
sendtime
msgtype

The *type_description* can be any *user_type_name* or a complete type description (see the earlier section in this chapter, "Type Declarations").

The first unspecified parameter in any operation statement is assumed be the **requestport** unless a **requestport** parameter was already specified. This is the port that the message is to be sent to. If a **replyport** parameter is specified, it will be used as the port that the reply message is sent to. If no **replyport** parameter is specified, a per-thread global port is used for the reply message.

The keywords **in**, **out**, and **inout** are optional and indicate the direction of the parameter. The keyword **in** is used with parameters that are to be sent to the server. The keyword **out** is used with parameters to be returned by the server. The keyword **inout** is used with parameters to be both sent and returned. If no such keyword is given, the default is **in**.

The keywords **waittime**, **replyport**, and **msgtype** can be used to specify dynamic values for the wait time, the reply port, or the message type for this message. These parameters aren't passed to the server code, but are used when generating the send and receive calls. The **requestport** and **replyport** parameters must be of types that resolve to MSG_TYPE_PORT. The **waittime** and **msgtype** parameters must resolve to MSG_TYPE_INTEGER_32.

The keyword **skip** is provided to allow a procedure to be removed from a subsystem without causing all the subsequent message interfaces to be renumbered. It causes no code to be generated, but uses up a **msg_id** number.

Here are some examples:

```
procedure init_seed (        server_port : port_t;
                             seed        : dbl);
routine get_random (         server_port : port_t;
                             out num     : int);
simpleroutine use_random (   server_port : port_t;
                             info_seed   : string80;
                             info        : comp_arr;
                             info_1      : words);
simpleprocedure exit (       server_port : port_t);
```

See the earlier section in this chapter, "Programming Example," for an example of a complete subsystem definition.

Options Declarations

Several special-purpose options about the generated code may be specified. Defaults are available for each, and simple interfaces don't usually need to change them. First-time readers may want to skip this section. These options may occur more than once in the specification file. Each time an option declaration appears, it sets that option for all the following operations.

The waittime Specification

The **waittime** specification has one of the following two forms:

waittime *time* **;**
nowaittime ;

The word **waittime** is followed by an integer or an identifier that specifies the maximum time in milliseconds that the user code will wait for a reply from the server. If an identifier is used, it should be declared as an **extern** variable by some module in the user code. If the **waittime** option is omitted, or if the **nowaittime** statement is seen, the RPC doesn't return until a message is received.

The timeout value for the **msg_receive()** can alternatively be controlled by using a **waittime** parameter to the RPC.

The sendtime Specification

The **sendtime** specification has one of the following two forms:

sendtime *time* **;**
nosendtime ;

The word **sendtime** is followed by an integer or an identifier that specifies the maximum time in milliseconds that the user code will wait for the number of messages queued on the server's port to fall below the port's backlog. If an identifier is used, it should be declared as an **extern** variable by some module in the user code. If the **sendtime** option is omitted, or if the **nosendtime** statement is seen, the RPC doesn't return until the message has been enqueued on the server's port.

The timeout value for the **msg_send()** can be controlled alternatively by using a **sendtime** parameter to the RPC.

The msgtype Specification

The **msgtype** specification has the following form:

msgtype *msgtype_value* **;**

msgtype_value may be one of the values from the header file **mach/msg_type.h**. The available types are MSG_TYPE_RPC and MSG_TYPE_NORMAL. The MSG_TYPE_RPC is set to a correct value by default; this value normally shouldn't be changed. The value MSG_TYPE_NORMAL can be used to reset the **msgtype** option.

The **msgtype** value for the **msg_send()** can be controlled alternatively by using a **msgtype** parameter to the RPC.

The error Specification

The **error** specification has the following form:

error *error_proc* ;

The **error** specification is used to specify how message-passing errors are to be handled for operations other than routines or simple routines. In all types of routines, any message errors are returned in the return value of the routine. For operations of types other than routines, the procedure *error_proc* is called when a message error is detected. The procedure specified by *error_proc* has to be supplied by the user, and must be of the form:

void *error_proc* (kern_return_t *error_code*)

If the **error** specification is omitted, *error_proc* is set to **MsgError()**.

The serverprefix Specification

The **serverprefix** specification has the following form:

serverprefix *string* ;

The word **serverprefix** is followed by an identifier string that will be prepended to the actual names of all the following server-side functions implementing the message operations. This is particularly useful when it's necessary for the user-side and server-side functions to have different names, as must be the case when a server is also a user of copies of itself.

The userprefix Specification

The **userprefix** specification has the following form:

userprefix *string* ;

The word **userprefix** is followed by an identifier string that will be prepended to the actual names of all the following user-side functions calling the message operations. **serverprefix** should usually be used when different names are needed for the user and server functions, but **userprefix** is also available for the sake of completeness.

The rcsid Specification

The **rcsid** specification has the following form:

rcsid *string* **;**

This specification causes a string variable *sys*_**user_rscid** in the user module and *sys*_**server_rcsid** in the server module to be set equal to the input string. The subsystem name *sys* was described earlier in this chapter, in the section "Subsystem Identification."

Syntax Summary

This section summarizes the syntax of MiG specification files. Note the following conventions:

- Terminal symbols (literals) are shown in boldface type.
- Nonterminal symbols are shown in italic type.
- Alternatives are listed on separate lines.
- Brackets indicate zero or one occurrence of the bracketed item. An ellipsis (...) indicates one or more repetitions of the preceding item. Brackets and ellipsis combined, as in [*item* ...] indicate zero, one, or more repetitions of the item.
- Types must be declared before they're used.
- Comments may be included in a ".defs" file if surrounded by **/*** and ***/**. Comments are parsed and removed by the C preprocessor.

specification_file:
subsystem_description [*waittime_description*] [*sendtime_description*]
[*msgtype_description*] [*error_description*] [*server_prefix_description*]
[*user_prefix_description*] [*rscid_description*] [*type_description* ...]
[*import_declaration* ...] *operation_description* ...

subsystem_description:
subsystem *identifier decimal_integer* **;**

waittime_description:
waittime *time_value* **;**
nowaittime ;

sendtime_description:
sendtime *time_value* **;**
nosendtime ;

time_value:
MSG_TYPE_INTEGER_32

msgtype_description:
msgtype *msgtype_value* **;**

msgtype_value:
MSG_TYPE_RPC
MSG_TYPE_NORMAL

error_description:
error *error_procedure* **;**

server_prefix_description:
serverprefix *identifier_string* **;**

user_prefix_description:
userprefix *identifier_string* **;**

rcsid_description:
rcsid *identifier_string* **;**

type_description:
type *type_definition* **;**

import_declaration:
import_keyword include_name **;**

import_keyword:
import
uimport
simport

include_name:
"*file_name*"
<*file_name*>

operation_description:
    routine_description
    simpleroutine_description
    procedure_description
    simpleprocedure_description
    function_description

routine_description:
    routine *argument_list* **;**

simpleroutine_description:
    simpleroutine *argument_list* **;**

procedure_description:
    procedure *argument_list* **;**

simpleprocedure_description:
    simpleprocedure *argument_list* **;**

function_description:
    function *argument_list* **:** *type_definition* **;**

argument_list:
    ([*argument_definition*] [**;** *argument_definition*] ... **)**

argument_definition:
    [*specification*] *identifier* **:** *type_definition* [**, dealloc**]

specification:
    in
    out
    inout
    requestport
    replyport
    waittime
    msgtype

type_definition:
    identifier **=** [^] [*repetition* ...] *ipc_info* [*translation*]

repetition:
    array [[*size*] **] of**
    struct [[*size*] **] of**

size:
    integer_expression

integer_expression:
    integer_expression **+** *integer_expression*
    integer_expression – *integer_expression*
    integer_expression * *integer_expression*
    integer_expression / *integer_expression*
    (*integer_expression*)
    integer

ipc_info:
    (*ipc_type_name* , *size_in_bits* [, **dealloc**])
    ipc_type_name
    identifier

translation:
    [*input_function*] [*output_function*] [*destructor_function*]

input_function:
    intran : *identifier*

output_function:
    outtran : *identifier*

destructor_function:
    destructor : *identifier*

ipc_type_name:
    integer
    manifest_constant

Compiling MiG Specification Files

To compile a MiG specification file, specify the name of your ".defs" file (or files) and any switches as arguments to the **mig** command. For example:

```
mig -v random.defs
```

MiG recognizes the following switches:

[**handler** *name*]
: Specifies a name for the file that's usually called *sys***Server.c**, and specifies that it should provide a handler interface instead of the usual server interface. An additional header file called *sys*_**handler.h** is also produced, as if the **sheader** option were specified. Handler interfaces are used mainly in loadable kernel servers; they're discussed in Chapter 6, "Designing Loadable Kernel Servers."

[**header** *name*]
: Specifies a name for the file that's usually called *sys***.h**.

[**p**,**P**]
: If **p**, use 2-byte message padding. You should use this option only if your server or client might be exchanging messages containing fields shorter than 4 bytes with a client or server that was built using NeXT Software Release 1. If **P**, use 4-byte message padding. The default value is **P**. For example, a 1-byte message element would be padded to 2 bytes if you specify **p**, or 4 bytes by default.

[**q**,**Q**]
: If **q**, suppress warning statements. If **Q**, print warning statements. The default value is **Q**.

[**r**,**R**]
: If **r**, use **msg_rpc**(); if **R**, use **msg_send**(), **msg_receive**() pairs. The default value is **r**.

[**s**,**S**]
: If **s**, generate symbol table with *sys***Server.c** code. The layout of a symbol table (**mig_symtab_t**) is defined in the header file **mach/mig_errors.h**. If **S**, suppress the symbol table. The default value is **S**. This is useful for protection systems where access to the server's operations is dynamically specifiable or for providing a run-time indirected server call interface with **syscall**() (server-to-server calls made on behalf of a client).

[**server** *name*]
: Specifies a name for the file that's usually called *sys***Server.c**.

[**sheader** *name*]
: Specifies that MiG create an additional header file, called *name*, that's suitable for inclusion in the server defined by the ".defs" file.

[**user** *name*]
: Specifies a name for the file that's usually called *sys***User.c**.

[**v**,**V**]
: If **v** (verbose), print routines and types as they're processed. If **V**, compile silently. The default value is **V**.

Any switches MiG doesn't recognize are passed to the C preprocessor. MiG also notices if the **-MD** option is being passed to the C preprocessor. If it is, MiG fixes up the resulting ".d" file to show the dependencies of the ".h," and ".c" files on the ".defs" file and any included ".defs" files. For this feature to work correctly, the name of the subsystem must be the same as the name of the ".defs" file.

MiG runs the C preprocessor to process comments and preprocessor macros such as **#include** or **#define**. For example, the following statement can be used to include the type definitions for standard Mach and C types:

```
#include <mach/std_types.defs>
```

The output from the C preprocessor is then passed to the program **migcom**, which generates the C files.

3 *Using Loadable Kernel Servers*

This chapter discusses how to use the kernel-server loader functions to interact with *loadable kernel servers*. Loadable kernel servers are modules, such as device drivers and network protocols, that can be added to the NeXT Mach kernel. One example of interaction with a loadable kernel server is using the function **kern_loader_load_server()** to load a loadable kernel server. Another example is using the function **kern_loader_server_list()** to get a list of all loadable kernel servers that are either loaded or prepared for loading (allocated).

The following section gives more information on loadable kernel servers and on the kernel-server loader, **kern_loader**. The next section gives some examples of using the kernel-server loader functions. Each kernel-server loader function is described in detail in Chapter 4, "Mach Functions."

For more information about loadable kernel servers, see Part 2 of this manual, "Writing Loadable Kernel Servers." Part 2 also has information about the kernel-server utility, **kl_util**, which is a command-line interface to many of the functions described in this chapter.

How Loadable Kernel Servers Work

The kernel-server loader is a server task that's automatically called during system startup. When started, it reads a list of loadable kernel servers out of its configuration file, **/etc/kern_loader.conf**, and allocates these servers.

A loadable kernel server is a module that's loaded into the kernel after the system has been booted. Because loadable kernel servers are the only way to add kernel functionality without recompiling the whole kernel, they're the only way for anyone outside of NeXT to write kernel-level device drivers and network protocols. However, third parties aren't the only ones to use loadable kernel servers—NeXT uses them for drivers of devices that many people won't have.

For example, the graphics tablet driver is a loadable kernel server that is loaded by the application **/NextAdmin/InstallTablet**. Having a loadable tablet driver is advantageous because performance on NeXT computers that don't have a graphics tablet (the majority of NeXT computers) is better than if the tablet driver were always in the kernel.

Loadable kernel servers can have three states:

- Allocated—The kernel-server loader (**kern_loader**) has allocated space and resources for the loadable kernel server and is listening for Mach messages to its ports. However, the server isn't currently loaded into the kernel.
- Loaded—The loadable kernel server is running.
- Unallocated—The kernel-server loader has no space or other resources allocated for the loadable kernel server.

Not all loadable kernel servers stay in the allocated state when they're initialized. Servers that don't use Mach messages, for example, are loaded immediately. Most message-based servers, however, stay in the allocated state until the kernel-server loader receives either a message for the server or a request such as **kern_loader_load_server()** that tells it to load the server.

Each loadable kernel server stays loaded until the kernel-server loader either shuts down or receives a request to unload or delete the server (such as **kern_loader_unload_server()**).

See Chapter 5, "Overview of Loadable Kernel Servers," for more information on loadable kernel servers and on using **kern_loader**.

Overview of Kernel-Server Loader Functions

This section describes the use of the kernel-server loader functions. See the "Kernel-Server Loader Functions" section of Chapter 4 for more information on each of the functions.

The kernel-server loader functions are broken into two groups—those that deal with a single loadable kernel server, and those that deal with the kernel-server loader itself. Two more functions help you print error messages: **kern_loader_error()** and **kern_loader_error_string()**.

Before you can call any other kernel-server loader function, you must call **kern_loader_look_up()** to obtain the port of the kernel-server loader. You must provide this port as an argument to all of the following function calls. A similar argument, returned by **kern_loader_server_com_port()**, is required only for calls to functions that deal with a server's message logging.

Use **kern_loader_add_server()** to cause a loadable kernel server to be allocated. If the server starts automatically, then it will be loaded into the kernel; otherwise, you can send a message to the server or call **kern_loader_load_server()** to load the server into the kernel. To remove a loadable kernel server from the kernel, use **kern_loader_unload_server()** (to leave the server in the allocated state) or **kern_loader_delete_server()** (to deallocate kernel-server loader resources for the server).

For each loadable kernel server, logging is off by default. To get status messages from a particular loadable kernel server, use **kern_loader_log_level()** to turn the server's logging on and **kern_loader_get_log()** to get the next log message. You might want to turn logging off (by again using **kern_loader_log_level()**) before you stop collecting log messages, since messages continue to be logged and take system space even when no one requests them.

You can get detailed information about the state of a particular server by calling **kern_loader_server_info()**.

Use **kern_loader_status_port()** to register a port where log messages from the kernel-server loader should be sent. These messages usually reflect changes in the state of one or more loadable kernel servers. You can get a list of all the servers that the kernel-server loader knows about by calling **kern_loader_server_list()**. Use **kern_loader_abort()** to shut down or reconfigure the kernel-server loader. Use **kern_loader_ping()** to make sure either that the kernel-server loader is responding normally to messages, or that all outstanding status messages have been sent.

Functions for Asynchronous Messages

Three of the kernel-server loader functions don't immediately return information. Instead, these three functions tell the kernel-server loader to send asynchronous reply messages that contain the information. Whenever you call one of these functions, you must supply the code necessary to handle the kernel-server loader's reply message. The following table shows the three asynchronous kernel-server loader functions and their corresponding user-written functions.

Asynchronous Function	User-Written Function
kern_loader_ping()	*ping_func()*
kern_loader_get_log()	*log_data_func()*
kern_loader_status_port()	*string_func()*

This section describes how to handle asynchronous reply messages from the kernel-server loader. First it describes the code that all three of the asynchronous functions require in your program. Then it describes how to implement the handler necessary for each of the functions.

Common Code for Handling Reply Messages

If your program calls a kernel-server loader function that sends an asynchronous reply message, then your program must follow these steps to handle reply messages:

1. Allocate a port on which to receive messages from the kernel-server loader.
2. Call the asynchronous function, passing as data the receiving port.
3. Listen to the receiving port (often in a separate thread).
4. After receiving a message on the port, call **kern_loader_reply_handler()**.
5. Take care of the reply message in a handling function, which is called by **kern_loader_reply_handler()**.

You must write the handling function that's called by **kern_loader_reply_handler()**. You must also create a structure that specifies which handling functions exist; you pass a pointer to this structure to **kern_loader_reply_handler()** every time you call it. The structure is of type **kern_loader_reply_t**, which is defined in the header file **kernserv/kern_loader_reply_handler.h** as the following:

```
typedef struct kern_loader_reply {
    void         *arg;        /* argument to pass to function */
    msg_timeout_t timeout;    /* timeout for RPC return msg_send */
    kern_return_t (*string)(  /* kern_loader_status_port() function */
                void         *arg,
                printf_data_t string,
                unsigned int  string_count,
                int           level);
    kern_return_t (*ping)(    /* kern_loader_ping() function */
                void      *arg,
                int        id);
    kern_return_t (*log_data)(/* kern_loader_get_log() function */
                void         *arg,
                printf_data_t log_data,
                unsigned int  log_data_count);
} kern_loader_reply_t;
```

The following example calls one of the asynchronous kernel-server loader functions, **kern_loader_status_port()**. The handler for the reply message is called **print_string()**, and is specified to the kernel-server loader using the structure **reply_handlers**.

```
#import <mach/mach.h>
#import <mach/mach_error.h>
#import <kernserv/kern_loader_types.h>
#import <kernserv/kern_loader.h>
#import <kernserv/kern_loader_reply_handler.h>
#import <kernserv/kern_loader_error.h>
#import <mach/cthreads.h>
#import <libc.h>
#import <stdio.h>

void receive_thread(port_name_t port);
kern_return_t print_string(void *arg, printf_data_t string,
    unsigned int string_count, int level);
```

```
void main()
{
    kern_return_t   r;
    port_name_t     status_port, kl_port;

    r = kern_loader_look_up(&kl_port);
    if (r != KERN_SUCCESS) {
        mach_error("kl_util: can't find kernel loader", r);
        exit(1);
    }

    r = port_allocate(task_self(), &status_port);
    if (r != KERN_SUCCESS) {
        mach_error("kl_util: can't allocate reply port", r);
        exit(1);
    }

    /* Get generic status messages on this port. */
    r = kern_loader_status_port(kl_port, status_port);
    if (r != KERN_SUCCESS) {
        kern_loader_error("Couldn't specify status port", r);
            exit(1);
    }

    /* Create a thread to listen on status_port. */
    cthread_detach(cthread_fork((cthread_fn_t)receive_thread,
        (any_t)status_port));

    /*
     * Sleep for a while so we can enter kl_util commands at a shell
     * window.  The output of all commands (except status lines from
     * kl_util -s) will show up in both the window that's running this
     * program and in the window that's running kl_util.  (kl_util
     * also has a status port registered.)
     */
    sleep(30);
    exit(0);
}

kern_loader_reply_t reply_handlers = {
    0,                  /* argument to pass to function */
    0,                  /* timeout for rpc return msg_send */
    print_string,       /* string function */
    0,                  /* reply_ping function */
    0                   /* log_data function */
};
```

```
void receive_thread(port_name_t port)
{
    char          msg_buf[kern_loader_replyMaxRequestSize];
    msg_header_t *msg = (msg_header_t *)msg_buf;
    kern_return_t r;

    /* message handling loop */
    while (TRUE) {
        /* Receive the next message in the queue. */
        msg->msg_size = kern_loader_replyMaxRequestSize;
        msg->msg_local_port = port;
        r = msg_receive(msg, MSG_OPTION_NONE, 0);
        if (r != KERN_SUCCESS)
            break;

        /* Handle the message we just received. */
        kern_loader_reply_handler(msg, &reply_handlers);
    }

    /* We get here only if msg_receive returned an error. */
    mach_error("receive_thread", r);
    exit(1);
}

/*
 * This function is called by kern_loader every time it has status to
 * report.
 */
kern_return_t print_string(void *arg, printf_data_t string,
    unsigned int string_count, int level)
{
    /* If the string is empty, return. */
    if (string_count == 0 || !string)
        return KERN_SUCCESS;

    /* Print the string we were passed, with our special prefix. */
    printf("print_string: %s", string);

    return KERN_SUCCESS;
}
```

Handling a Status Message

You can receive many reply messages as the result of just one call to **kern_loader_status_port()**. The function you must use to handle these reply messages is defined as follows:

kern_return_t *string_func*(void **arg*, printf_data_t *string*, u_int *string_count*, int *level*)

The first argument, *arg*, has the same value as the **arg** field in the **kern_loader_reply_t** structure. The string that the kernel-server loader is logging is returned in *string*, with the string's length returned in *string_count*. The *level* argument is set to the priority of the log message, using the priorities defined in the header file **sys/syslog.h** (LOG_EMERG, LOG_ALERT, and so on).

Your function should return KERN_SUCCESS.

The following code is an example of a *string_func* named **print_string()**.

```
/*
 * This function is called by kern_loader every time it has status to
 * report.
 */
kern_return_t print_string(void *arg, printf_data_t string,
    u_int string_count, int level)
{
    /* If the string is empty, return. */
    if (string_count == 0 || !string)
        return KERN_SUCCESS;

    /* Print the string we were passed, with our special prefix. */
    printf("print_string: %s", string);

    return KERN_SUCCESS;
}
```

Handling a Synchronization Message

A call to **kern_loader_ping()** results in a single reply message. Your handler for this reply message must have the following syntax:

kern_return_t *ping_func*(void **arg*, int *id*)

The first argument, *arg*, is the value in the **arg** field of the **kern_loader_reply_t** structure. *id* is the same as the *id* value specified in the call to **kern_loader_ping()**. Your *ping_func* should return KERN_SUCCESS.

Here's an example of a *ping_func* that causes its task to shut down.

```
/* This function is called after a kern_loader_ping(). */
kern_return_t ping (void *arg, int id)
{
    exit(0);    /* Kill this process. */
}
```

Handling a Log Message

Each time you call **kern_loader_log_data()**, you receive a single reply message as soon as any log data from the specified driver is available. The function you write to handle this message must have the following syntax:

kern_return_t *log_func*(void **arg*, printf_data_t *log_data*,
unsigned int *log_data_count*)

The first argument has the same value as the *arg* field in the **kern_loader_reply_t** structure. The *log_data* argument is a string containing the log entry from the loadable kernel server, preceded by a time stamp that indicates the relative time when the kernel-server loader received the log message. The *log_data_count* is the size of *log_data* in bytes. You should call **vm_deallocate()** on *log_data* when it's no longer needed.

Your *log_func* should return KERN_SUCCESS.

Here's an example of a *log_func* called **log_data**. It prints out the log message it's passed, and then requests another log message.

```
kern_return_t log_data(void *arg, printf_data_t log_data,
    unsigned int log_data_count)
{
    kern_return_t r;

    /* Print the string we were passed, with our prefix. */
    printf("log_data: %s\n", log_data);
    vm_deallocate(task_self(), (vm_address_t)log_data,
        log_data_count*sizeof(*log_data));

    /* Request the next log message. */
    r = kern_loader_get_log(kl_port, server_com_port, reply_port);
    if (r != KERN_SUCCESS) {
        kern_loader_error("log_data:  Error calling
            kern_loader_get_log", r);
        exit(1);
    }

    return KERN_SUCCESS;
}
```

4 *Mach Functions*

This chapter gives detailed descriptions of the C functions provided by the NeXT Mach operating system. It also describes some macros that behave like functions. For this chapter, the functions and macros are divided into five groups:

- C-thread functions—Use these to implement multiple threads in an application.
- Mach kernel functions—Use these to get access to the Mach operating system.
- Bootstrap Server functions—Use these to set up communication between the task that provides a local service and the tasks that use the service.
- Network Name Server functions—Use these to set up communication between tasks that might not be on the same machine.
- Kernel-server loader functions—Use these to load and unload loadable kernel servers, to add and delete servers to and from the kernel-server loader, and to get information about servers.

Within each section, functions are subgrouped with other functions that perform related tasks. These subgroups are described in alphabetical order by the name of the first function listed in the subgroup. Functions within subgroups are also listed alphabetically, with a pointer to the subgroup description.

For convenience, these functions are summarized in the *NeXTSTEP Programming Interface Summary.* The summary lists functions by the same subgroups used in this chapter and combines several related subgroups under a heading such as "Basic C-Thread Functions" or "Task Functions." For each function, the summary shows the calling sequence.

C-Thread Functions

These functions provide a C language interface to the low-level, language-independent primitives for manipulating threads of control.

In a multithreaded application, you should use the C-thread functions whenever possible, rather than Mach kernel functions. If you need to call a Mach kernel function that requires a **thread_t** argument, you can find the Mach thread that corresponds to a particular C thread by calling **cthread_thread**().

---

condition_alloc(), mutex_alloc()

SUMMARY Create a condition or mutex object

SYNOPSIS

#import <mach/cthreads.h>

condition_t **condition_alloc**(void)
mutex_t **mutex_alloc**(void)

DESCRIPTION The macros **condition_alloc**() and **mutex_alloc**() provide dynamic allocation of condition and mutex objects. When you're finished using these objects, you can deallocate them using **condition_free**() and **mutex_free**().

EXAMPLE

```
my_condition = condition_alloc();
my_mutex = mutex_alloc();
```

SEE ALSO **condition_init**(), **mutex_init**(), **condition_free**(), **mutex_free**()

condition_broadcast()

SUMMARY Broadcast a condition

SYNOPSIS **#import <mach/cthreads.h>**

void **condition_broadcast**(condition_t *c*)

DESCRIPTION The macro **condition_broadcast()** wakes up all threads that are waiting (with **condition_wait()**) for the condition *c*. This macro is similar to **condition_signal()**, except that **condition_signal()** doesn't wake up every waiting thread.

EXAMPLE

```
any_t listen(any_t arg)
{
    mutex_lock(my_mutex);
    while(!data)
        condition_wait(my_condition, my_mutex);
    /* . . . */
    mutex_unlock(my_mutex);

    mutex_lock(printing);
    printf("Condition has been met\n");
    mutex_unlock(printing);
}

main()
{
    my_condition = condition_alloc();
    my_mutex = mutex_alloc();
    printing = mutex_alloc();

    cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)0));

    mutex_lock(my_mutex);
    data = 1;
    mutex_unlock(my_mutex);
    condition_broadcast(my_condition);
    /* . . . */
}
```

SEE ALSO **condition_signal()**, **condition_wait()**

condition_clear(), mutex_clear()

SUMMARY Clear a condition or mutex object

SYNOPSIS **#import <mach/cthreads.h>**

void **condition_clear**(struct condition *c)
void **mutex_clear**(struct mutex *m)

DESCRIPTION You must call one of these macros before freeing an object of type **struct condition** or **struct mutex**. See the discussion of **condition_init()** and **mutex_init()** for information on why you might want to use these types instead of **condition_t** and **mutex_t**.

EXAMPLE

```
struct mystruct {
    my_data_t     data;
    struct mutex  m;
};
struct mystruct  *mydata;
mydata = (struct mystruct *)malloc(sizeof (struct mystruct));

mutex_init(&mydata->m);
/* . . . */
mutex_lock(&mydata->m);
/* Do something to mydata that only one thread can do. */
mutex_unlock(&mydata->m);
/* . . . */
mutex_clear(&mydata->m);
free(mydata);
```

SEE ALSO **condition_init(), mutex_init(), condition_free(), mutex_free()**

---

condition_free(), mutex_free()

SUMMARY Deallocate a condition or mutex object

SYNOPSIS **#import <mach/cthreads.h>**

void **condition_free**(condition_t *c*)
void **mutex_free**(mutex_t *m*)

DESCRIPTION The macros **condition_free()** and **mutex_free()** let you deallocate condition and mutex objects that were allocated dynamically. Before deallocating such an object, you must guarantee that no other thread will reference it. In particular, a thread blocked in **mutex_lock()** or **condition_wait()** should be viewed as referencing the object continually; freeing the object out from under such a thread is erroneous, and can result in bugs that are extremely difficult to track down.

SEE ALSO **condition_alloc(), mutex_alloc(), condition_clear(), mutex_clear()**

---

condition_init(), mutex_init()

SUMMARY Initialize a condition variable or mutex

SYNOPSIS **#import <mach/cthreads.h>**

void **condition_init**(struct condition *_c_)
void **mutex_init**(struct mutex *_m_)

DESCRIPTION The macros **condition_init()** and **mutex_init()** initialize an object of the **struct condition** or **struct mutex** referent type, so that its address can be used wherever an object of type **condition_t** or **mutex_t** is expected. Initialization of the referent type is most often used when you have included the referent type itself (rather than a pointer) in a larger structure, for more efficient storage allocation.

For instance, a data structure might contain a component of type **struct mutex** to allow each instance of that structure to be locked independently. During initialization of the instance, you would call **mutex_init()** on the **struct mutex** component. The alternative of using a **mutex_t** component and initializing it using **mutex_alloc()** would be less efficient.

If you're going to free a condition or mutex object of type **struct condition** or **struct mutex**, you should first clear it using **condition_clear()** or **mutex_clear()**.

EXAMPLE

```
struct mystruct {
    my_data_t     data;
    struct mutex  m;
};
struct mystruct  *mydata;
mydata = (struct mystruct *)malloc(sizeof (struct mystruct));

mutex_init(&mydata->m);
/* . . . */
mutex_lock(&mydata->m);
/* Do something to mydata that only one thread can do. */
mutex_unlock(&mydata->m);
/* . . . */
mutex_clear(&mydata->m);
free(mydata);
```

SEE ALSO **condition_alloc(), mutex_alloc(), condition_clear(), mutex_clear()**

---

condition_name(), condition_set_name(), mutex_name(), mutex_set_name()

SUMMARY Associate a string with a condition or mutex variable

SYNOPSIS

#import <mach/cthreads.h>

char ***condition_name**(condition_t *c*)
void **condition_set_name**(condition_t *c*, char **name*)
char ***mutex_name**(mutex_t *m*)
void **mutex_set_name**(mutex_t *m*, char **name*)

DESCRIPTION These macros let you associate a name with a condition or a mutex object. The name is used when trace information is displayed. You can also use this name for your own application-dependent purposes.

EXAMPLE

```
/* Do something if this is a "TYPE 1" condition. */
if (strcmp(condition_name(c), "TYPE 1") == 0)
    /* Do something. */;
```

condition_set_name() → See condition_name()

condition_signal()

SUMMARY Signal a condition

SYNOPSIS **#import <mach/cthreads.h>**

void **condition_signal**(condition_t *c*)

DESCRIPTION The macro **condition_signal()** should be called when one thread needs to indicate that the condition represented by the condition variable is now true. If any other threads are waiting (using **condition_wait()**), at least one of them will be awakened. If no threads are waiting, nothing happens. The macro **condition_broadcast()** is similar to this one, except that it wakes up *all* threads that are waiting.

EXAMPLE

```
any_t listen(any_t arg)
{
    mutex_lock(my_mutex);
    while(!data)
        condition_wait(my_condition, my_mutex);
    /* . . . */
    mutex_unlock(my_mutex);

    mutex_lock(printing);
    printf("Condition has been met\n");
    mutex_unlock(printing);
}

main()
{
    my_condition = condition_alloc();
    my_mutex = mutex_alloc();
    printing = mutex_alloc();

    cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)0));

    mutex_lock(my_mutex);
    data = 1;
    mutex_unlock(my_mutex);
    condition_signal(my_condition);
    /* . . . */
}
```

SEE ALSO **condition_broadcast(), condition_wait()**

condition_wait()

SUMMARY Wait on a condition

SYNOPSIS **#import <mach/cthreads.h>**

void **condition_wait**(condition_t *c*, mutex_t *m*)

DESCRIPTION The function **condition_wait()** unlocks the mutex it takes as a argument, suspends the calling thread until the specified condition is likely to be true, and locks the mutex again when the thread resumes. There's no guarantee that the condition will be true when the thread resumes, so this function should always be used as follows:

```
mutex_t m;
condition_t c;

mutex_lock(m);
/* . . . */
while    (/* condition isn't true */)
    condition_wait(c, m);
/* . . . */
mutex_unlock(m);
```

SEE ALSO **condition_broadcast()**, **condition_signal()**

cthread_abort()

SUMMARY Interrupt a C thread

SYNOPSIS **#import <mach/cthreads.h>**

kern_return_t **cthread_abort**(cthread_t *t*)

DESCRIPTION This function provides the functionality of **thread_abort()** to C threads. The **cthread_abort()** function interrupts system calls; it's usually used along with **thread_suspend()**, which stops a thread from executing any more user code. Calling **cthread_abort()** on a thread that isn't suspended is risky, since it's difficult to know exactly

what system trap, if any, the thread might be executing and whether an interrupt return would cause the thread to do something useful.

See **thread_abort()** for a full description of the use of this function.

---

cthread_count()

SUMMARY Get the number of threads in this task

SYNOPSIS **#import <mach/cthreads.h>**

int **cthread_count()**

DESCRIPTION This function returns the number of threads that exist in the current task. You can use this function to help make sure that your task doesn't create too many threads (over 200 or so). See **cthread_set_limit()** for information on restricting the number of threads in a task.

EXAMPLE

```
printf("C thread count should be 1, is %d\n", cthread_count());
cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)0));
printf("C thread count should be 2, is %d\n", cthread_count());
```

SEE ALSO **cthread_limit()**, **cthread_set_limit()**

---

cthread_data(), cthread_set_data()

SUMMARY Associate data with a thread

SYNOPSIS **#import <mach/cthreads.h>**

any_t **cthread_data**(cthread_t *t*)
void **cthread_set_data**(cthread_t *t*, any_t *data*)

DESCRIPTION The macros **cthread_data()** and **cthread_set_data()** let you associate arbitrary data with a thread, providing a simple form of thread-specific "global" variable. More elaborate mechanisms, such as per-thread property lists or hash tables, can then be built with these macros.

EXAMPLE

```
int listen(any_t arg)
{
    mutex_lock(printing);
    printf("This thread's data is: %d\n",
        (int)cthread_data(cthread_self()));
    mutex_unlock(printing);
    /* . . . */
}

main()
{
    cthread_t lthread;

    printing = mutex_alloc();

    lthread = cthread_fork((cthread_fn_t)listen, (any_t)0);
    cthread_set_data(lthread, (any_t)100);
    cthread_detach(lthread);
    /* . . . */
}
```

SEE ALSO **cthread_name(), cthread_set_name()**

---

cthread_detach()

SUMMARY Detach a thread

SYNOPSIS **#import <mach/cthreads.h>**

void **cthread_detach**(cthread_t *t*)

DESCRIPTION The function **cthread_detach()** is used to indicate that **cthread_join()** will never be called on the given thread. This is usually known at the time the thread is forked, so the most efficient usage is the following:

cthread_detach(cthread_fork(*function*, *argument*));

A thread may, however, be detached at any time after it's forked, as long as no other attempt is made to join it or detach it.

EXAMPLE

```
cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)reply_port));
```

SEE ALSO **cthread_fork(), cthread_join()**

---

cthread_errno()

SUMMARY Get a thread's errno value

SYNOPSIS

#import <mach/cthreads.h>

int **cthread_errno**(void)

DESCRIPTION Use the **cthread_errno()** function to get the errno value for the current thread. In the UNIX operating system, **errno** is a process-wide global variable that's set to an error number when a UNIX system call fails. However, because Mach has multiple threads per process, Mach keeps errno information on a per-thread basis as well as in **errno**.

Like the value of **errno**, the value returned by **cthread_errno()** is valid only if the last UNIX system call returned –1. Errno values are defined in the header file **bsd/sys/errno.h**.

EXAMPLE

```
int ret;

ret = chown(FILEPATH, newOwner, newGroup);
if (ret == -1) {
    if (cthread_errno() == ENAMETOOLONG)
        /* . . . */
}
```

SEE ALSO **cthread_set_errno(), intro(2)** UNIX manual page

---

cthread_exit()

SUMMARY Exit a thread

SYNOPSIS

#import <mach/cthreads.h>

void **cthread_exit**(any_t *result*)

DESCRIPTION The function **cthread_exit()** terminates the calling thread. The result is passed to the thread that joins the caller, or is discarded if the caller is detached.

An implicit **cthread_exit()** occurs when the top-level function of a thread returns, but it may also be called explicitly.

EXAMPLE

```
cthread_exit(0);
```

SEE ALSO **cthread_detach(), cthread_fork(), cthread_join()**

---

cthread_fork()

SUMMARY Fork a thread

SYNOPSIS **#import <mach/cthreads.h>**

cthread_t **cthread_fork**(any_t (**function*)(), any_t *arg*)

DESCRIPTION The function **cthread_fork()** takes two arguments: a function for the new thread to execute, and an argument to this function. The **cthread_fork()** function creates a new thread of control in which the specified function is executed concurrently with the caller's thread. This is the sole means of creating new threads.

The **any_t** type represents a pointer to any C type. The **cthread_t** type is an integer-size handle that uniquely identifies a thread of control. Values of type **cthread_t** will be referred to as thread identifiers. Arguments larger than a pointer must be passed by reference. Similarly, multiple arguments must be simulated by passing a pointer to a structure containing several components. The call to **cthread_fork()** returns a thread identifier that can be passed to **cthread_join()** or **cthread_detach()**. Every thread must be either joined or detached exactly once.

EXAMPLE

```
cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)reply_port));
```

SEE ALSO **cthread_detach(), cthread_exit(), cthread_join()**

cthread_join()

SUMMARY Join threads

SYNOPSIS

#import <mach/cthreads.h>

any_t **cthread_join**(cthread_t *t*)

DESCRIPTION The function **cthread_join()** suspends the caller until the specified thread *t* terminates. The caller receives either the result of *t*'s top-level function or the argument with which *t* explicitly called **cthread_exit()**.

Attempting to join one's own thread results in deadlock.

EXAMPLE

```
cthread_t t;
t = cthread_fork((any_t (*)())listen, (any_t)reply_port);
/* . . . (Do some work, perhaps forking other threads.) */
result = cthread_join(t);  /* Wait for the thread to finish executing. */
/* . . . (Continue doing work) */
```

SEE ALSO **cthread_detach()**, **cthread_exit()**, **cthread_fork()**

---

cthread_limit(), cthread_set_limit()

SUMMARY Get or set the maximum number of threads in this task

SYNOPSIS

#import <mach/cthreads.h>

int **cthread_limit**(void)
void **cthread_set_limit**(int *limit*)

ARGUMENTS *limit*: The new maximum number of C threads per task. Specify zero if you want no limit.

DESCRIPTION These functions can help you to avoid creating too many threads. The danger in creating a large number of threads is that the kernel might run out of resources and panic. Usually, a task should avoid creating more than about 200 threads.

Use **cthread_set_limit()** to set a limit on the number of threads in the current task. When the limit is reached, new C threads will appear to fork successfully. However, they will have no associated Mach thread, so they won't do anything.

Use **cthread_limit()** to find out how many threads can exist in the current task. If the returned value is zero (the default), then no limit is currently being enforced.

Important: Use **cthread_count()** to determine when your task is approaching the maximum number of threads.

EXAMPLE

```
cthread_set_limit(LIMIT);

/* . . . */

/* Fork if we haven't reached the limit. */
if ( (LIMIT == 0) || (LIMIT > cthread_count()) )
    cthread_detach(cthread_fork((any_t (*)())a_thread,(any_t)0));
```

---

cthread_name(), cthread_set_name()

SUMMARY Associate a string with a thread

SYNOPSIS

#import <mach/cthreads.h>

char ***cthread_name**(cthread_t *t*)
void **cthread_set_name**(cthread_t *t*, char **name*)

DESCRIPTION The functions **cthread_name()** and **cthread_set_name()** let you associate an arbitrary name with a thread. The name is used when trace information is displayed. The name may also be used for application-specific diagnostics.

EXAMPLE

```
int listen(any_t arg)
{
    mutex_lock(printing);
    printf("This thread's name is: %s\n",
        cthread_name(cthread_self()));
    mutex_unlock(printing);
    /* . . . */
}
```

```
main()
{
    cthread_t lthread;

    printing = mutex_alloc();

    lthread = cthread_fork((cthread_fn_t)listen, (any_t)0);
    cthread_set_name(lthread, "lthread");
    cthread_detach(lthread);
    /* . . . */
}
```

SEE ALSO **cthread_data(), cthread_set_data()**

---

cthread_priority(), cthread_max_priority()

SUMMARY Set the scheduling priority for a C thread

SYNOPSIS **#import <mach/cthreads.h>**

kern_return_t **cthread_priority**(cthread_t *t*, int *priority*, boolean_t *set_max*)
kern_return_t **cthread_max_priority**(cthread_t *t*, processor_set_t *processor_set*, int *max_priority*)

ARGUMENTS *t*: The C thread whose priority is to be changed.

priority: The new priority to change it to.

set_max: Also set *t*'s maximum priority if true.

processor_set: The privileged port for the processor set to which *thread* is currently assigned.

max_priority: The new maximum priority.

DESCRIPTION These functions give C threads the functionality of **thread_priority()** and **thread_max_priority()**. See those functions for more details than are provided here.

The **cthread_priority()** function changes the base priority and (optionally) the maximum priority of *t*. If the new base priority is higher than the scheduled priority of the currently executing thread, this thread might be preempted. The maximum priority of the thread is also set if *set_max* is true. This call fails if *priority* is greater than the current maximum priority of the thread. As a result, **cthread_priority()** can lower—but never raise—the value of a thread's maximum priority.

The **cthread_max_priority()** function changes the maximum priority of the thread. Because it requires the privileged port for the processor set, this call can reset the maximum priority to any legal value. If the new maximum priority is less than the thread's base priority, then the thread's base priority is set to the new maximum priority.

EXAMPLE

```
/* Get the privileged port for the default processor set. */
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error!=KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

/* Set the max priority. */
error=cthread_max_priority(cthread_self(), default_set_priv,
    priority);
if (error!=KERN_SUCCESS)
    mach_error("Call to cthread_max_priority() failed",error);

/* Set the thread's priority. */
error=cthread_priority(cthread_self(), priority, FALSE);
if (error!=KERN_SUCCESS)
    mach_error("Call to cthread_priority() failed",error);
```

RETURN KERN_SUCCESS: Operation completed successfully

KERN_INVALID_ARGUMENT: *cthread* is not a C thread, *processor_set* is not a privileged port for a processor set, or *priority* is out of range (not in 0-31).

KERN_FAILURE: The requested operation would violate the thread's maximum priority (only for **cthread_priority**()) or the thread is not assigned to the processor set whose privileged port was presented.

SEE ALSO **thread_priority**(), **thread_max_priority**(), **thread_policy**(), **task_priority**(), **processor_set_priority**()

---

cthread_self()

SUMMARY Return the caller's C-thread identifier

SYNOPSIS **#import <mach/cthreads.h>**

cthread_t **cthread_self**(void)

DESCRIPTION The function **cthread_self**() returns the caller's own C-thread identifier, which is the same value that was returned by **cthread_fork**() to the creator of the thread. The C-thread identifier uniquely identifies the thread, and hence may be used as a key in data structures that associate user data with individual threads. Since thread identifiers may be reused by the underlying implementation, you should be careful to clean up such associations when threads exit.

EXAMPLE

```
printf("This thread's name is: %s\n",
    cthread_name(cthread_self()));
mutex_unlock(printing);
```

SEE ALSO **cthread_fork**(), **cthread_thread**(), **thread_self**()

cthread_set_data() → See cthread_data()

cthread_set_errno_self()

SUMMARY Set the current thread's errno value

SYNOPSIS **#import <mach/cthreads.h>**

void **cthread_set_errno_self**(int *error*)

DESCRIPTION Use this function to set the errno value for the current thread to *error*. In the UNIX operating system, **errno** is a process-wide global variable that's set to an error number when a UNIX system call fails. However, because Mach has multiple threads per process, Mach keeps errno information on a per-thread basis as well as in **errno**. This function has no effect on the value of **errno**.

The current thread's errno value can be obtained by calling **cthread_errno**(). Errno values are defined in the header file **bsd/sys/errno.h**.

EXAMPLE

```
cthread_set_errno_self(EPERM);
```

SEE ALSO **cthread_errno**(), **intro(2)** UNIX manual page

cthread_set_limit() → See cthread_limit()

cthread_set_name() → See cthread_name()

cthread_thread()

SUMMARY Return the caller's Mach thread identifier

SYNOPSIS **#import <mach/cthreads.h>**

thread_t **cthread_thread**(cthread_t *t*)

DESCRIPTION The macro **cthread_thread**() returns the Mach thread that corresponds to the specified C thread *t*.

EXAMPLE

```
/* Save the cthread and thread values for the forked thread. */
l_cthread = cthread_fork((cthread_fn_t)listen, (any_t)0);
cthread_detach(l_cthread);
l_realthread = cthread_thread(l_cthread);
```

SEE ALSO **cthread_fork(), cthread_self()**

---

cthread_yield()

SUMMARY Yield the processor to other threads

SYNOPSIS **#import <mach/cthreads.h>**

void **cthread_yield**(void)

DESCRIPTION The function **cthread_yield()** is a hint to the scheduler, suggesting that this would be a convenient point to schedule another thread to run on the current processor.

EXAMPLE

```
int i, n;

/* n is set previously */
for (i = 0; i < n; i += 1)
    cthread_yield();
```

SEE ALSO **cthread_priority(), thread_switch()**

mutex_alloc() → See condition_alloc()

mutex_clear() → See condition_clear()

mutex_free() → See condition_free()

mutex_init() → See condition_init()

mutex_lock()

SUMMARY Lock a mutex variable

SYNOPSIS **#import <mach/cthreads.h>**

void **mutex_lock**(mutex_t *m*)

DESCRIPTION The macro **mutex_lock()** attempts to lock the mutex *m* and blocks until it succeeds. If several threads attempt to lock the same mutex concurrently, one will succeed, and the others will block until *m* is unlocked. A deadlock occurs if a thread attempts to lock a mutex it has already locked.

EXAMPLE

```
/* Only one thread at a time should call printf. */
mutex_lock(printing);
printf("Condition has been met\n");
mutex_unlock(printing);
```

SEE ALSO **mutex_try_lock()**, **mutex_unlock()**

mutex_name() → See condition_name()

mutex_set_name() → See condition_name()

mutex_try_lock()

SUMMARY Try to lock a mutex variable

SYNOPSIS **#import <mach/cthreads.h>**

int **mutex_try_lock**(mutex_t *m*)

DESCRIPTION The function **mutex_try_lock()** attempts to lock the mutex *m*, like **mutex_lock()**, and returns true if it succeeds. If *m* is already locked, however, **mutex_try_lock()** immediately returns false rather than blocking. For example, a busy-waiting version of **mutex_lock()** could be written using **mutex_try_lock()**:

```
void mutex_lock(mutex_t m)
{
    for (;;)
        if (mutex_try_lock(m))
            return;
}
```

SEE ALSO **mutex_lock()**, **mutex_unlock()**

---

mutex_unlock()

SUMMARY Unlock a mutex variable

SYNOPSIS **#import <mach/cthreads.h>**

void **mutex_unlock**(mutex_t *m*)

DESCRIPTION The function **mutex_unlock()** unlocks *m*, giving other threads a chance to lock it.

EXAMPLE

```
/* Only one thread at a time should call printf. */
mutex_lock(printing);
printf("Condition has been met\n");
mutex_unlock(printing);
```

SEE ALSO **mutex_lock()**, **mutex_try_lock()**

exc_server()

SUMMARY Dispatch a message received on an exception port

SYNOPSIS

#import <mach/mach.h>
#import <mach/exception.h>

boolean_t **exc_server**(msg_header_t *_in_, msg_header_t *_out_)

ARGUMENTS _in_: A message that was received on the exception port. This message structure should be at least 64 bytes long.

out: An empty message to be filled by **exc_server()** and then sent. This message buffer should be at least 32 bytes long.

DESCRIPTION This function calls the appropriate exception handler. You should call this function after you've received a message on an exception port that you set up previously. Usually, this function is used along with a user-defined exception handler, which must have the following protocol:

kern_return_t **catch_exception_raise**(port_t *exception_port*, port_t *thread*, port_t *task*, int *exception*, int *code*, int *subcode*)

To receive a message on an exception port, you must first create a new port and make it the task or thread exception port. (You can't use the default task exception port because you can't get receive rights for it.) Before calling **msg_receive()**, you must set the **local_port** field of the header to the appropriate exception port and the **msg_size** field to the size of the structure for the incoming message.

If it accepted the incoming message, **exc_server()** returns true; otherwise it returns false.

You should keep a global value that indicates whether your exception handler successfully handled the exception. If it couldn't, then you should forward the exception message to the old exception port.

EXAMPLE

```
typedef struct {
    port_t old_exc_port;
    port_t clear_port;
    port_t exc_port;
} ports_t;

volatile boolean_t  pass_on = FALSE;
mutex_t             printing;

/* Listen on the exception port. */
any_t exc_thread(ports_t *port_p)
{
    kern_return_t   r;
    char           *msg_data[2][64];
    msg_header_t   *imsg = (msg_header_t *)msg_data[0],
                   *omsg = (msg_header_t *)msg_data[1];

    /* Wait for exceptions. */
    while (1) {
        imsg->msg_size = 64;
        imsg->msg_local_port = port_p->exc_port;
        r = msg_receive(imsg, MSG_OPTION_NONE, 0);

        if (r==RCV_SUCCESS) {
            /* Give the message to the Mach exception server. */
            if (exc_server(imsg, omsg)) {
                /* Send the reply message that exc_serv gave us. */
                r = msg_send(omsg, MSG_OPTION_NONE, 0);
                if (r != SEND_SUCCESS) {
                    mach_error("msg_send", r);
                    exit(1);
                }
            }
            else { /* exc_server refused to handle imsg. */
                mutex_lock(printing);
                printf("exc_server didn't like the message\n");
                mutex_unlock(printing);
                exit(2);
            }
        }
        else { /* msg_receive() returned an error. */
                mach_error("msg_receive", r);
                exit(3);
        }
```

```
        /* Pass the message to old exception handler, if necessary. */
        if (pass_on == TRUE) {
            imsg->msg_remote_port = port_p->old_exc_port;
            imsg->msg_local_port = port_p->clear_port;
            r = msg_send(imsg, MSG_OPTION_NONE, 0);
            if (r != SEND_SUCCESS) {
                mach_error("msg_send to old_exc_port", r);
                exit(4);
            }
        }
    }
}

/*
 * catch_exception_raise() is called by exc_server().  The only
 * exception it can handle is EXC_SOFTWARE.
 */
kern_return_t catch_exception_raise(port_t exception_port,
    port_t thread, port_t task, int exception, int code, int subcode)
{
    if ((exception == EXC_SOFTWARE) && (code == 0x20000)) {
        /* Handle the exception so that the program can continue. */
        mutex_lock(printing);
        printf("Handling the exception\n");
        mutex_unlock(printing);
        return KERN_SUCCESS;
    }
    else { /* Pass the exception on to the old port. */
        pass_on = TRUE;
        mach_NeXT_exception("Forwarding exception", exception,
            code, subcode);
        return KERN_FAILURE;  /* Couldn't handle this exception. */
    }
}

main()
{
    int             i;
    kern_return_t   r;
    ports_t         ports;

    printing = mutex_alloc();
```

```
    /* Save the old exception port for this task. */
    r = task_get_exception_port(task_self(), &(ports.old_exc_port));
    if (r != KERN_SUCCESS) {
        mach_error("task_get_exception_port", r);
        exit(1);
    }

    /* Create a new exception port for this task. */
    r = port_allocate(task_self(), &(ports.exc_port));
    if (r != KERN_SUCCESS) {
        mach_error("port_allocate 0", r);
        exit(1);
    }
    r = task_set_exception_port(task_self(), (ports.exc_port));
    if (r != KERN_SUCCESS) {
        mach_error("task_set_exception_port", r);
        exit(1);
    }

    /* Fork the thread that listens to the exception port. */
    cthread_detach(cthread_fork((cthread_fn_t)exc_thread,
        (any_t)&ports));
    /* Raise the exception. */
    ports.clear_port = thread_self();
    r = exception_raise(ports.exc_port, thread_reply(),
        ports.clear_port, task_self(), EXC_SOFTWARE, 0x20000, 6);

    if (r != KERN_SUCCESS)
        mach_error("catch_exception_raise didn't handle exception",
            r);
    else {
        mutex_lock(printing);
        printf("Successfully called exception_raise\n");
        mutex_unlock(printing);
    }
}
```

SEE ALSO **exception_raise(), mach_NeXT_exception()**

exception_raise()

SUMMARY Cause an exception to occur

SYNOPSIS

#import <mach/mach.h>
#import <mach/exception.h>

kern_return_t **exception_raise**(port_t *exception_port*, port_t *clear_port*, port_t *thread*, port_t *task*, int *exception*, int *code*, int *subcode*)

ARGUMENTS

exception_port: The exception port of the affected thread. (If the thread doesn't have its own exception port, then this should be the exception port of the task.)

clear_port: The port to which a reply message should be sent from the exception handler. If you don't care to see the reply, you can use **thread_reply**().

thread: The thread in which the exception condition occurred. If the exception isn't thread-specific, then specify THREAD_NULL.

task: The task in which the exception condition occurred.

exception: The type of exception that occurred; for example, EXC_SOFTWARE. Values for this variable are defined in the header file **mach/exception.h**.

code: The exception code. The meaning of this code depends on the value of *exception*.

subcode: The exception subcode. The meaning of this subcode depends on the values of *exception* and *code*.

DESCRIPTION

This function causes an exception message to be sent to *exception_port*, which results in a call to the exception handler. Usually this function is used along with a user-defined exception handler. (See **exc_server()** and **mach_NeXT_exception()** for more information on user-defined exception handlers.)

You can obtain *exception_port* by calling **thread_get_exception_port()** or (if no thread exception port exists or the exception affects the whole task) **task_get_exception_port()**.

If you're defining your own type of exception, you must have *exception* equal to EXC_SOFTWARE and *code* equal to or greater than 0x20000.

EXAMPLE

```
/* Raise the exception. */
r = exception_raise(ports.exc_port, thread_reply(), thread_self(),
    task_self(), EXC_SOFTWARE, 0x20000, 6);
if (r != KERN_SUCCESS)
    mach_error("catch_exception_raise didn't handle exception", r);
else {
    /* Use mutex so only one thread at a time can call printf. */
    mutex_lock(printing);
    printf("Successfully called exception_raise\n");
    mutex_unlock(printing);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_FAILURE: The exception handler didn't successfully deal with the exception.

KERN_INVALID_ARGUMENT: One of the arguments wasn't valid.

SEE ALSO **exc_server(), mach_NeXT_exception(), task_get_exception_port(), thread_get_exception_port()**

---

host_info()

SUMMARY Get information about a host

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **host_info**(host_t *host*, int *flavor*, host_info_t *host_info*, unsigned int **host_info_count*)

ARGUMENTS

host: The host for which information is to be obtained.

flavor: The type of statistics to be returned. Currently HOST_BASIC_INFO, HOST_PROCESSOR_SLOTS, and HOST_SCHED_INFO are implemented.

host_info: Returns statistics about *host*.

host_info_count: The number of integers in the info structure; returns the number of integers that Mach tried to fill the info structure with. For HOST_BASIC_INFO, you should set *host_info_count* to HOST_BASIC_INFO_COUNT. For HOST_PROCESSOR_SLOTS, you should set it to the maximum number of CPUs (returned by HOST_BASIC_INFO). For HOST_SCHED_INFO, set it to HOST_SCHED_INFO_COUNT.

DESCRIPTION Returns the selected information array for a host, as specified by *flavor*. The *host_info* argument is an array of integers that's supplied by the caller and returned filled with specified information. The *host_info_count* argument is supplied by the caller as the maximum number of integers in *host_info* (which can be larger than the space required for the information). On return, it contains the actual number of integers in *host_info*.

Warning: This replaces the old **host_info()** call. It isn't backwards compatible.

Basic information is defined by HOST_BASIC_INFO. Its size is defined by HOST_BASIC_INFO_COUNT. Possible values of the **cpu_type** and **cpu_subtype** fields are defined in the header file **mach/machine.h**, which is included in **mach/mach.h**.

```
struct host_basic_info {
    int           max_cpus;     /* maximum possible cpus for
                                 * which kernel is configured */
    int           avail_cpus;   /* number of cpus now available */
    vm_size_t     memory_size;  /* size of memory in bytes */
    cpu_type_t    cpu_type;     /* cpu type */
    cpu_subtype_t cpu_subtype;  /* cpu subtype */
};
typedef struct host_basic_info *host_basic_info_t;
```

Processor slots of the active (available) processors are defined by HOST_PROCESSOR_SLOTS. The size of this information should be obtained from the **max_cpus** field of the structure returned by HOST_BASIC_INFO. HOST_PROCESSOR_SLOTS returns an array of integers, each of which is the slot number of a CPU.

Additional information of interest to schedulers is defined by HOST_SCHED_INFO. The size of this information is defined by HOST_SCHED_INFO_COUNT.

```
struct host_sched_info {
    int  min_timeout;  /* minimum timeout in milliseconds */
    int  min_quantum;  /* minimum quantum in milliseconds */
};

typedef struct host_sched_info *host_sched_info_t
```

EXAMPLE An example of using HOST_BASIC_INFO:

```
kern_return_t           ret;
struct host_basic_info  basic_info;
unsigned int            count=HOST_BASIC_INFO_COUNT;

ret=host_info(host_self(), HOST_BASIC_INFO,
    (host_info_t)&basic_info, &count);
if (ret != KERN_SUCCESS)
    mach_error("host_info() call failed", ret);
else printf("This system has %d bytes of RAM.\n",
    basic_info.memory_size);
```

An example of using HOST_PROCESSOR_SLOTS (you also need to include the HOST_BASIC_INFO code above so you can get **max_cpus**):

```
host_info_t  slots;
unsigned int cpu_count, i;

cpu_count=basic_info.max_cpus;
slots=(host_info_t)malloc(cpu_count*sizeof(int));
ret=host_info(host_self(), HOST_PROCESSOR_SLOTS, slots,
    &cpu_count);
if (ret!=KERN_SUCCESS)
    mach_error("PROCESSOR host_info() call failed", ret);
else for (i=0; i<cpu_count; i++)
    printf("CPU %d is in slot %d.\n", i, *slots++);
```

An example of using HOST_SCHED_INFO:

```
kern_return_t           ret;
struct host_sched_info  sched_info;
unsigned int            sched_count=HOST_SCHED_INFO_COUNT;

ret=host_info(host_self(), HOST_SCHED_INFO,
    (host_info_t)&sched_info, &sched_count);
if (ret != KERN_SUCCESS)
    mach_error("SCHED host_info() call failed", ret);
else
    printf("The minimum quantum is %d milliseconds.\n",
        sched_info.min_quantum);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host* is not a host, *flavor* is not recognized, or (for HOST_PROCESSOR_SLOTS) **count* is less than **max_cpus**.

KERN_FAILURE: **count* is less than HOST_BASIC_INFO_COUNT (when *flavor* is HOST_BASIC_INFO) or HOST_SCHED_INFO_COUNT (for HOST_SCHED_INFO).

MIG_ARRAY_TOO_LARGE: Returned info array is too large for *host_info*. The *host_info* argument is filled as much as possible, and *host_info_count* is set to the number of elements that would be returned if there were enough room.

SEE ALSO **host_kernel_version(), host_processors(), processor_info()**

---

host_kernel_version()

SUMMARY Get kernel version information

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **host_kernel_version**(host_t *host*, kernel_version_t *version*)

ARGUMENTS *host*: The host for which information is being requested.

version: Returns a character string describing the kernel version executing on *host*.

DESCRIPTION This function returns the version string compiled into *host*'s kernel at the time it was built. If you don't use the **kernel_version_t** declaration, then you should allocate KERNEL_VERSION_MAX bytes for the version string.

EXAMPLE

```
kern_return_t       ret;
kernel_version_t    string;

ret=host_kernel_version(host_self(), string);
if (ret != KERN_SUCCESS)
    mach_error("host_kernel_version() call failed", ret);
else
    printf("Version string:  %s\n", string);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host* was not a host.

KERN_INVALID_ADDRESS: *version* points to inaccessible memory.

SEE ALSO **host_info()**, **host_processors()**, **processor_info()**

host_priv_self() → See host_self()

---

host_processor_set_priv()

SUMMARY Get the privileged port of a processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **host_processor_set_priv**(host_priv_t *host_priv*,
processor_set_t *processor_set_name*, processor_set_t **processor_set*)

ARGUMENTS *host_priv*: The privileged host port for the desired host.

processor_set_name: The name port of the processor set.

processor_set: Returns the privileged port of the processor set.

DESCRIPTION This function returns send rights to the privileged port for the specified processor set. This port is used in calls that can affect other threads or tasks. For example, **processor_set_tasks()** requires the privileged port because it returns the port of every task on the system.

EXAMPLE

```
kern_return_t    error;
processor_set_t  processor_set;
processor_set_t  default_set;

error=processor_set_default(host_self(), &default_set);
if (error != KERN_SUCCESS)
    mach_error("Call to processor_set_default failed", error);

error=host_processor_set_priv(host_priv_self(), default_set,
    &processor_set);
if (error != KERN_SUCCESS)
    mach_error("Call to host_processor_set_priv failed; make sure
        you're superuser", error);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host_priv* was not a privileged host port, or *processor_set_name* didn't name a valid processor set.

---

host_processor_sets()

SUMMARY

Get the name ports of all processor sets on a host

SYNOPSIS

#import <mach/mach.h>

kern_return_t **host_processor_sets**(host_t *host*,
processor_set_name_array_t **processor_set_list*, unsigned int **processor_set_count*)

ARGUMENTS

host: The host port for the desired host.

processor_set_list: Returns an array of processor sets currently existing on *host*; no particular ordering is guaranteed.

processor_set_ count: Returns the number of processor sets in the *processor_set_list.*

DESCRIPTION

This function returns send rights to the name port for each processor set currently assigned to *host*. The **host_processor_set_priv()** function can be used to obtain the privileged ports from these if desired. The *processor_set_list* argument is an array that is created as a result of this call. You should call **vm_deallocate()** on this array when the data is no longer needed.

Note: In single-processor systems, you can get the same information by calling **processor_set_default()**.

EXAMPLE

```
kern_return_t                ret;
processor_set_name_array_t   list;
unsigned int                 count;

ret=host_processor_sets(host_self(), &list, &count);
if (ret!=KERN_SUCCESS)
    mach_error("error calling host_processor_sets", ret);
else {
    /* . . . */
    ret=vm_deallocate(task_self(), (vm_address_t)list,
        sizeof(list)*count);
    if (ret!=KERN_SUCCESS)
        mach_error("error calling vm_deallocate", ret);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host* is not a host.

SEE ALSO

host_processor_set_priv(), processor_set_create(), processor_set_tasks(), processor_set_threads(), processor_set_default()

---

host_processors()

SUMMARY

Get the processor ports for a host

SYNOPSIS

#import <mach/mach.h>

kern_return_t **host_processors**(host_priv_t *host_priv*, processor_array_t **processor_list*, unsigned int **processor_count*)

ARGUMENTS

host_priv: Privileged host port for the desired host.

processor_list: Returns the processors existing on *host_priv*; no particular ordering is guaranteed.

processor_count: Returns the number of processors in *processor_list*.

DESCRIPTION **host_processors()** gets send rights to the processor port for each processor existing on *host_priv*. The *processor_list* argument is an array that is created as a result of this call. The caller may wish to call **vm_deallocate()** on this array when the data is no longer needed.

EXAMPLE

```
kern_return_t      error;
processor_array_t  list;
unsigned int       count;

error=host_processors(host_priv_self(), &list, &count);
if (error!=KERN_SUCCESS){
    mach_error("error calling host_processors", error);
    exit(1);
}
/* . . . */
vm_deallocate(task_self(), (vm_address_t)list, sizeof(list)*count);
if (error!=KERN_SUCCESS)
    mach_error("Trouble freeing list", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host_priv* is not a privileged host port.

SEE ALSO **processor_info()**, **processor_start()**, **processor_exit()**, **processor_control()**

---

host_self(), host_priv_self()

SUMMARY Get the host port for this host

SYNOPSIS **#import <mach/mach.h>**

host_t **host_self**(void)
host_priv_t **host_priv_self**(void)

DESCRIPTION The **host_self()** function returns send rights to the host port for the host on which the call is executed. This port can be used only to obtain information about the host, not to control the host.

The **host_priv_self()** function returns send rights to the privileged host port for the host on which the call is executed. This port is used to control physical resources on that host and is only available to privileged tasks. PORT_NULL is returned if the invoker is not the UNIX superuser.

EXAMPLE

```
/* Get the privileged port for the default processor set. */
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error!=KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}
```

SEE ALSO **host_processors(), host_info(), host_kernel_version()**

---

mach_error(), mach_error_string()

SUMMARY Display or get a Mach error string

SYNOPSIS

#import <mach/mach.h>
#import <mach/mach_error.h>

void **mach_error**(char **string*, kern_return_t *error*)
char ***mach_error_string**(kern_return_t *error*)

ARGUMENTS *string*: The string you want displayed before the Mach error string.

error: The error value for which you want an error string.

DESCRIPTION The function **mach_error()** displays a message on **stderr**. The message contains the string specified by *string*, the string returned by **mach_error_string()**, and the actual error value (*error*). Since **mach_error()** isn't thread-safe, you might want to protect it with a mutex if you call it in a multiple-thread task.

The function **mach_error_string()** returns the string associated with *error.*

Note that because the error value specified by *error* is of type **kern_return_t**, these functions work only with Mach functions.

EXAMPLE

```
mutex_t        printing;

main()
{
    kern_return_t  error;
    port_t         result;

    printing = mutex_alloc();

    /* . . . */
    if ((error=port_allocate(task_self(), &result)) != KERN_SUCCESS) {
        mutex_lock(printing);
        mach_error("Error calling port_allocate", error);
        mutex_unlock(printing);
        exit(1);
    }
    /* . . . */
}
```

---

mach_NeXT_exception(), mach_NeXT_exception_string()

SUMMARY Display or get a Mach exception string

SYNOPSIS

#import <mach/mach.h>

void **mach_NeXT_exception**(char **string*, int *exception*, int *code*, int *subcode*)
char ***mach_NeXT_exception_string**(int *exception*, int *code*, int *subcode*)

ARGUMENTS

string: The string you want displayed before the Mach exception string.

exception: The exception value for which you want a string.

code: The exception code. How this is used depends on the value of *exception.*

subcode: The exception subcode. How this is used depends on the value of *exception.*

DESCRIPTION The function **mach_NeXT_exception()** displays a message on **stderr**. The message contains the string specified by *string*, then the string returned by **mach_NeXT_exception_string()**, and then the values of *exception*, *code*, and *subcode*. Since **mach_NeXT_exception()** isn't thread-safe, you might want to protect it with a mutex if you call it in a multiple-thread task.

The function **mach_NeXT_exception_string()** returns the string associated with *exception*, *code*, and *subcode*.

EXAMPLE

```
/*
 * catch_exception_raise() is called by exc_server().  The only
 * exception it can handle is EXC_SOFTWARE.
 */
kern_return_t catch_exception_raise(port_t exception_port,
    port_t thread, port_t task, int exception, int code, int subcode)
{
    if ((exception == EXC_SOFTWARE) && (code == 0x20000)) {
        /* Handle the exception so that the program can continue. */
        mutex_lock(printing);
        printf("Handling the exception\n");
        mutex_unlock(printing);
        return KERN_SUCCESS;
    }
    else { /* Pass the exception on to the old port. */
        pass_on = TRUE;
        mach_NeXT_exception("Forwarding exception", exception,
            code, subcode);
        return KERN_FAILURE;  /* Couldn't handle this exception. */
    }
}
```

SEE ALSO **exception_raise(), exc_server()**

---

map_fd()

SUMMARY Map a file into virtual memory

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **map_fd**(int *fd*, vm_offset_t *offset*, vm_offset_t **address*, boolean_t *find_space*, vm_size_t *size*)

ARGUMENTS *fd*: An open UNIX file descriptor for the file that's to be mapped.

offset: The byte offset within the file, at which mapping is to begin.

address: A pointer to an address in the calling process at which the mapped file should start. This address, unlike the offset, must be page-aligned.

find_space: If true, the kernel will select an unused address range at which to map the file and return its value in *address*.

size: The number of bytes to be mapped.

DESCRIPTION The function **map_fd()** is a UNIX extension that's technically not part of Mach. This function causes *size* bytes of data starting at *offset* in the file specified by *fd* to be mapped into the virtual memory at the address specified by *address*. If *find_space* is true, the input value of *address* can be null, and the kernel will find an unused piece of virtual memory to use. (You should free this space with **vm_deallocate()** when you no longer need it.) If you provide a value for *address*, it must be page-aligned and at least *size* bytes long. The sum of *offset* and *size* must not exceed the length of the file.

Memory mapping doesn't cause I/O to take place. When specific pages are first referenced, they cause page faults that bring in the data. The mapped memory is copy-on-write. Modified data is returned to the file only by a **write()** call.

EXAMPLE

```
kern_return_t r;
int           fd;
char         *memfile, *filename = "/tmp/myfile";

/* Open the file. */
fd = open(filename, O_RDONLY);

/* Map part of it into memory. */
r = map_fd(fd, (vm_offset_t)0, &(vm_offset_t)memfile, TRUE,
    (vm_size_t)5);
if (r != KERN_SUCCESS)
    mach_error("Error calling map_fd()", r);
else
    printf("Second character in %s is:  %c\n", filename, memfile[1]);
```

RETURN KERN_SUCCESS: The data was mapped successfully.

KERN_INVALID_ADDRESS: *address* wasn't valid.

KERN_INVALID_ARGUMENT: An invalid argument was passed.

msg_receive()

SUMMARY Receive a message

SYNOPSIS

#import <mach/mach.h>
#import <mach/message.h>

msg_return_t **msg_receive**(msg_header_t **header*, msg_option_t *option*,
msg_timeout_t *timeout*)

ARGUMENTS *header*: The address of a buffer in which the message is to be received. Two fields of the message header must be set before the call is made: **msg_local_port** must be set to the value of the port from which the message is to be received, and **msg_size** must be set to the maximum size of the message that may be received. This maximum size must be less than or equal to the size of the buffer.

option: The failure conditions under which **msg_receive()** should terminate. The value of this argument is a combination (using the bitwise OR operator) of the following options. Unless one of these values is explicitly specified, **msg_receive()** does not return until a message has been received.

RCV_TIMEOUT: Specifies that **msg_receive()** should return when the specified timeout elapses if a message has not arrived by that time; if not specified, the timeout will be ignored (that is, it will be infinite).

RCV_INTERRUPT: Specifies that **msg_receive()** should return when a software interrupt occurs in this thread.

RCV_LARGE: Specifies that **msg_receive()** should return without dequeuing a message if the next message in the queue is larger than *header*.**msg_size**. (Normally, a message that is too large is dequeued and lost.) You can use this option to dynamically determine how large your message buffer must be.

Use MSG_OPTION_NONE to specify that none of the above options is desired.

timeout: If RCV_TIMEOUT is specified in *option*, then *timeout* is the maximum time in milliseconds to wait for a message before giving up.

DESCRIPTION The function **msg_receive()** retrieves the next message from the port or port set specified in the **msg_local_port** field of *header*. If a port is specified, the port must not be a member of a port set.

If a port set is specified, then **msg_receive()** will retrieve messages sent to any of the set's member ports. Mach sets the **msg_local_port** field to the specific port on which the message was found. It's not an error for the port set to have no members, or for members to be added and removed from a port set while a **msg_receive()** on the port set is in progress.

The message consists of its header, followed by a variable amount of data; the message header supplied to **msg_receive()** must specify (in **msg_size**) the maximum size of the message that can be received into the buffer provided.

If no messages are present on the port(s) in question, **msg_receive()** will wait until a message arrives, or until one of the specified termination conditions is met (see the description of the *option* argument for this function).

If the message is successfully received, then **msg_receive()** sets the **msg_size** field of the header to the size of the received message. If the RCV_LARGE option was set and **msg_receive()** returned RCV_TOO_LARGE, then the **msg_size** field is set to the size of the message that was too large.

If the received message contains out-of-line data (that is, data for which the **msg_type_inline** attribute was specified as false), the data will be returned in a newly allocated region of memory; the message body will contain a pointer to that new region. You should deallocate this memory when the data is no longer needed. See the **vm_allocate()** call for a description of the state of newly allocated memory.

See Chapter 2, "Using Mach Messages," for information on setting up messages and on writing Mach servers.

EXAMPLE

```
msg_header_t   *imsg, header;

/* Wait for messages. */
while (1) {
    /* Set up the message structure. */
    header.msg_size = sizeof header;
    header.msg_local_port = receive_port;

    /* Get the next message on the queue. */
    r = msg_receive(&header, RCV_LARGE, 0);
```

```
    /* If the message is too big ... */
    if (r==RCV_TOO_LARGE) {
        /* ... allocate a structure for it ... */
        imsg = (msg_header_t *)malloc(header.msg_size);
        /* ... initialize the structure ... */
        imsg->msg_size = header.msg_size;
        imsg->msg_local_port = receive_port;
        /* ... and get the message. */
        r = msg_receive(imsg, MSG_OPTION_NONE, 0);
    }

    if (r==RCV_SUCCESS) {
        /* Handle the message. */
    }
    else { /* msg_receive() returned an error. */
        mach_error("msg_receive", r);
        exit(3);
    }
}
```

RETURN RCV_SUCCESS: The message has been received.

RCV_INVALID_MEMORY: The message specified was not writable by the calling task.

RCV_INVALID_PORT: An attempt was made to receive on a port to which the calling task does not have the proper access, or which was deallocated (see **port_deallocate()**) while waiting for a message.

RCV_TOO_LARGE: The message header and body combined are larger than the size specified by **msg_size**. Unless the RCV_LARGE option was set, the message has been dequeued and lost. If the RCV_LARGE option was specified, then Mach sets **msg_size** to the size of the message that was too large and leaves the message at the head of the queue.

RCV_NOT_ENOUGH_MEMORY: The message to be received contains more out-of-line data than can be allocated in the receiving task.

RCV_TIMED_OUT: The message was not received after *timeout* milliseconds.

RCV_INTERRUPTED: A software interrupt occurred and the RCV_INTERRUPT option was specified.

RCV_PORT_CHANGE: The port specified was added to a port set during the duration of the **msg_receive()** call.

msg_rpc()

SUMMARY Send and receive a message

SYNOPSIS

```
#import <mach/mach.h>
#import <mach/message.h>
```

msg_return_t **msg_rpc**(msg_header_t ***header*, msg_option_t *option*, msg_size_t *rcv_size*, msg_timeout_t *send_timeout*, msg_timeout_t *rcv_timeout*)

ARGUMENTS *header*: Address of a message buffer that will be used for both **msg_send()** and **msg_receive()**. This buffer contains a message header followed by the data for the message to be sent. The **msg_remote_port** field specifies the port to which the message is to be sent. The **msg_local_port** field specifies the port on which a message is then to be received; if this port is the special value PORT_DEFAULT, it gets replaced by the value PORT_NULL for the purposes of the **msg_send()** operation.

option: A union of the *option* arguments for the send and receive (see **msg_send()** and **msg_receive()**).

rcv_size: The maximum size allowed for the received message; this must be less than or equal to the size of the message buffer. The **msg_size** field in the header specifies the size of the message to be sent.

send_timeout, *rcv_timeout*: The timeout values to be applied to the component operations. These are used only if the option SEND_TIMEOUT or RCV_TIMEOUT is specified.

DESCRIPTION The function **msg_rpc()** is a hybrid call that performs a **msg_send()** followed by a **msg_receive()**, using the same message buffer. Because of the order of the send and receive, this function is appropriate for clients of Mach servers. However, the **msg_rpc()** call to a Mach server is usually performed by MiG-generated code, not by handwritten code.

See Chapter 2, "Using Mach Messages," for information on setting up messages and on writing Mach servers.

RETURN RPC_SUCCESS: The message was successfully sent and a reply was received.

Other possible values are the same as those for **msg_send()** and **msg_receive()**; any error during the **msg_send()** portion will terminate the call.

msg_send()

SUMMARY Send a message

SYNOPSIS

#import <mach/mach.h>
#import <mach/message.h>

msg_return_t **msg_send**(msg_header_t **header*, msg_option_t *option*,
msg_timeout_t *timeout*)

ARGUMENTS *header*: The address of the message to be sent. A message consists of a fixed-size header followed by a variable number of data descriptors and data items. See the header file **mach/message.h** for a definition of the message structure.

option: The failure conditions under which **msg_send()** should terminate. The value of this argument is a combination (using the bitwise OR operator) of the following options. Unless one of these values is explicitly specified, **msg_send()** does not return until the message is successfully queued for the intended receiver.

SEND_TIMEOUT: Specifies that the **msg_send()** request should terminate after the timeout period has elapsed, even if the kernel has been unable to queue the message.

SEND_NOTIFY: Allows the sender to send exactly one message without being suspended even if the destination port is full. When that message can be posted to the receiving port queue, this task receives a message that notifies it that another message can be sent. If the sender tries to send a second message with this option to the same port before the first notification arrives, the result is an error. If both SEND_NOTIFY and SEND_TIMEOUT are specified, **msg_send()** will wait until the specified timeout has elapsed before invoking the SEND_NOTIFY option.

SEND_INTERRUPT: Specifies that **msg_send()** should return if a software interrupt occurs in this thread.

Use MSG_OPTION_NONE to specify that none of the above options is wanted.

timeout: If the destination port is full and the SEND_TIMEOUT option has been specified, this value specifies the maximum wait time (in milliseconds).

DESCRIPTION The function **msg_send()** transmits a message from the current task to the port specified in the message header field. The message consists of its header, followed by a variable number of data descriptors and data items.

If the **msg_local_port** field isn't set to PORT_NULL, send rights to that port will be passed to the receiver of this message. The receiver task can use that port to send a reply to this message.

If the SEND_NOTIFY option is used and this call returns a SEND_WILL_NOTIFY code, you can expect to receive a notify message from the kernel. This message will be either a NOTIFY_MSG_ACCEPTED or a NOTIFY_PORT_DELETED message, depending on what happened to the queued message. The **notify_port** field in these messages is the port to which the original message was sent. The formats for these messages are defined in the header file **sys/notify.h**.

See Chapter 2, "Using Mach Messages," for information on setting up messages and on writing Mach servers.

EXAMPLE

```
/* From the handwritten part of a Mach server... */
while (TRUE)
{
    /* Receive a request from a client. */
    msg.head.msg_local_port = port;
    msg.head.msg_size = sizeof(struct message);
    ret = msg_receive(&msg.head, MSG_OPTION_NONE, 0);
    if (ret != RCV_SUCCESS) /* ignore errors */;

    /* Feed the request into the server. */
    (void)add_server(&msg, &reply);

    /* Send a reply to the client. */
    reply.head.msg_local_port = port;
    ret = msg_send(&reply.head, MSG_OPTION_NONE, 0);
    if (ret != SEND_SUCCESS) /* ignore errors */;
}
```

RETURN

SEND_SUCCESS: The message has been queued for the destination port.

SEND_INVALID_MEMORY: The message header or body was not readable by the calling task, or the message body specified out-of-line data that was not readable.

SEND_INVALID_PORT: The message refers either to a port for which the current task does not have access, or to which access was explicitly removed from the current task (see **port_deallocate**()) while waiting for the message to be posted, or a **msg_type_name** field in the message specifies rights that the name doesn't denote in the task (for example, specifying MSG_TYPE_SEND and supplying a port set name).

SEND_TIMED_OUT: The message was not sent since the destination port was still full after *timeout* milliseconds.

SEND_WILL_NOTIFY: The destination port was full but the SEND_NOTIFY option was specified. A notification message will be sent when the message can be posted.

SEND_NOTIFY_IN_PROGRESS: The SEND_NOTIFY option was specified but a notification request is already outstanding for this thread and given destination port.

---

port_allocate()

SUMMARY Create a port

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_allocate**(task_t *task*, port_name_t **port_name*)

ARGUMENTS *task*: The task in which the new port is created (for example, use **task_self()** to specify the caller's task).

port_name: Returns the name used by *task* for the new port.

DESCRIPTION The function **port_allocate()** causes a port to be created for the specified task; the resulting port is returned in *port_name*. The target task initially has both send and receive rights to the port. The new port isn't a member of any port set.

EXAMPLE

```
port_t          myport;
kern_return_t   error;

if ((error=port_allocate(task_self(), &myport)) != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: A port has been allocated.

KERN_INVALID_ARGUMENT: *task* was invalid.

KERN_RESOURCE_SHORTAGE: No more port slots are available for this task.

SEE ALSO **port_deallocate()**

port_deallocate()

SUMMARY Deallocate a port

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_deallocate**(task_t *task*, port_name_t *port_name*)

ARGUMENTS *task*: The task that wants to relinquish rights to the port (for example, use **task_self()** to specify the caller's task).

port_name: The name that *task* uses for the port to be deallocated.

DESCRIPTION The function **port_deallocate()** requests that the target task's access to a port be relinquished.

If *task* has receive rights for the port and the port doesn't have a backup port, these things happen:

- The port is destroyed.
- All other tasks with send access to the port are notified of its destruction.
- If the port is a member of a port set, it's removed from the port set.

If *task* has receive rights for the port and the port *does* have a backup port, then the following things happen:

- If the port is a member of a port set, it's removed from the port set.
- Send and receive rights for the port are sent to the backup port in a notification message (see **port_set_backup()**).

EXAMPLE

```
port_t          my_port;
kern_return_t   error;

/* . . . */

error=port_deallocate(task_self(), my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_deallocate failed", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: The port has been deallocated.

KERN_INVALID_ARGUMENT: *task* was invalid or *port_name* doesn't name a valid port.

SEE ALSO **port_allocate()**

---

port_extract_receive(), port_extract_send()

SUMMARY Remove access rights to a port and return them to the caller

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_extract_receive**(task_t *task*, port_name_t *its_name*, port_t **its_port*)
kern_return_t **port_extract_send**(task_t *task*, port_name_t *its_name*, port_t **its_port*)

ARGUMENTS *task*: The task whose rights the caller takes.

its_name: The name by which *task* knows the port.

its_port: Returns the receive or send rights.

DESCRIPTION The functions **port_extract_receive()** and **port_extract_send()** remove the port access rights that *task* has for a port and return the rights to the caller. This leaves *task* with no rights for the port.

The **port_extract_send()** function extracts send rights; *task* can't have receive rights for the named port. The **port_extract_receive()** function extracts receive rights.

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid or *its_name* doesn't name a port for which *task* has the required rights.

SEE ALSO **port_insert_send()**, **port_insert_receive()**

port_extract_send() → See port_extract_receive()

port_insert_receive(), port_insert_send()

SUMMARY Give a task rights with a specific name

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_insert_receive**(task_t *task*, port_t *my_port*, port_name_t *its_name*)
kern_return_t **port_insert_send**(task_t *task*, port_t *my_port*, port_name_t *its_name*)

ARGUMENTS *task*: The task getting the new rights.

my_port: Rights supplied by the caller.

its_name: The name by which *task* will know the new rights.

DESCRIPTION The functions **port_insert_receive()** and **port_insert_send()** give a task rights with a specific name. If *task* already has rights named *its_name*, or has some other name for *my_port*, the operation will fail. The *its_name* argument can't be a predefined port, such as PORT_NULL.

The **port_insert_send()** function inserts send rights, and **port_insert_receive()** inserts receive rights.

RETURN KERN_SUCCESS: The call succeeded.

KERN_NAME_EXISTS: *task* already has a right named *its_name*.

KERN_FAILURE: *task* already has rights to *my_port*.

KERN_INVALID_ARGUMENT: *task* was invalid or *its_name* was an invalid name.

SEE ALSO **port_extract_send(), port_extract_receive()**

port_insert_send() → See port_insert_receive()

---

port_names()

SUMMARY Get information about the port name space of a task

SYNOPSIS

#import <mach/mach.h>

kern_return_t **port_names**(task_t *task*, port_name_array_t **port_names*,
unsigned int **port_names_count*, port_type_array_t **port_types*,
unsigned int **port_types_count*)

ARGUMENTS *task*: The task whose port name space is queried.

port_names: Returns the names of the ports and port sets in the port name space of *task*, in no particular order.

port_names_count: Returns the number of names returned.

port_types: Returns the type of each corresponding name. This indicates what kind of rights the task holds for the port, or whether the name refers to a port set. The type is one of the following: PORT_TYPE_SEND (send rights only), PORT_TYPE_RECEIVE_OWN (send and receive rights), PORT_TYPE_SET (the port is a port set).

port_types_count: Returns the same value as *port_names_count*.

DESCRIPTION The function **port_names()** returns information about the port name space of *task*. It returns the port and port set names that are currently valid for *task*. For each name, it also returns what type of rights *task* holds.

The *port_names* and *port_types* arguments are arrays that are automatically allocated when the reply message is received. You should use **vm_deallocate()** on them when the data is no longer needed.

EXAMPLE

```
kern_return_t       error;
port_name_array_t   names;
unsigned int        names_count, types_count;
port_type_array_t   types;

error=port_names(task_self(), &names, &names_count, &types,
    &types_count);
if (error != KERN_SUCCESS) {
    mach_error("port_rename returned value of ", error);
    exit(1);
}
/* . . . */
error=vm_deallocate(task_self(), (vm_address_t)names,
    sizeof(names)*names_count);
if (error != KERN_SUCCESS)
    mach_error("Trouble freeing names", error);

error=vm_deallocate(task_self(), (vm_address_t)types,
    sizeof(names)*types_count);
if (error != KERN_SUCCESS)
    mach_error("Trouble freeing types", error);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid.

SEE ALSO

port_type(), port_status(), port_set_status()

---

port_rename()

SUMMARY

Change the name by which a port or port set is known to a task

SYNOPSIS

#import <mach/mach.h>

kern_return_t **port_rename**(task_t *task*, port_name_t *old_name*, port_name_t *new_name*)

ARGUMENTS *task*: The task whose port name space is changed.

old_name: The current name of the port or port set.

new_name: The new name for the port or port set.

DESCRIPTION The function **port_rename()** changes the name by which a port or port set is known to *task*. The port name specified in *new_name* must not already be in use, and it can't be a predefined port, such as PORT_NULL. Currently, a name is a small integer.

One way to guarantee that a name isn't already in use is to deallocate a port and then use its name as *new_name*. Another way is to check all the existing names, using **port_names()**, before you call **port_rename()**. If you choose another naming scheme, you should be prepared to try another name if **port_rename()** returns a KERN_NAME_EXISTS error.

EXAMPLE

```
#define MY_PORT  (port_name_t)99

port_name_t       my_port;
kern_return_t     error;

error=port_allocate(task_self(),&my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}

error=port_rename(task_self(), my_port, MY_PORT);
if (error == KERN_NAME_EXISTS)
    /* try again with a different name */;
else if (error != KERN_SUCCESS) {
        mach_error("port_rename failed", error);
        exit(1);
    }
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_NAME_EXISTS: *task* already has a port or port set named *new_name*.

KERN_INVALID_ARGUMENT: *task* was invalid, or *task* didn't know any ports or port sets named *old_name*, or *new_name* was an invalid name.

SEE ALSO **port_names()**

port_set_add()

SUMMARY Move the named port into the named port set

SYNOPSIS

#import <mach/mach.h>

kern_return_t **port_set_add**(task_t *task*, port_set_name_t *set_name*,
port_name_t *port_name*)

ARGUMENTS

task: The task that has receive rights for the port set and port.

set_name: *task*'s name for the port set.

port_name: *task*'s name for the port.

DESCRIPTION The function **port_set_add()** moves the named port into the named port set. The *task* must have receive rights for the port. If the port is already a member of another port set, it's removed from that set first.

EXAMPLE

```
kern_return_t    error;
port_set_name_t  set_name;
port_t           my_port;

error=port_set_allocate(task_self(),&set_name);
if (error != KERN_SUCCESS) {
    mach_error("port_set_allocate failed", error);
    exit(1);
}

error=port_allocate(task_self(),&my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}

error=port_set_add(task_self(), set_name, my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_NOT_RECEIVER: *task* doesn't have receive rights for the port.

KERN_INVALID_ARGUMENT: *task* was invalid, or *set_name* doesn't name a valid port set, or *port_name* doesn't name a valid port.

SEE ALSO **port_set_remove()**

---

port_set_allocate()

SUMMARY Create a port set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_set_allocate**(task_t *task*, port_set_name_t **set_name*)

ARGUMENTS *task*: The task in which the new port set is created.

set_name: Returns *task*'s name for the new port set.

DESCRIPTION The function **port_set_allocate()** causes a port set to be created for the specified task; the resulting set's name is returned in *set_name*. The new port set is empty.

EXAMPLE

```
kern_return_t    error;
port_set_name_t  set_name;

error=port_set_allocate(task_self(),&set_name);
if (error != KERN_SUCCESS) {
    mach_error("port_set_allocate failed", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid.

KERN_RESOURCE_SHORTAGE: The kernel ran out of memory.

SEE ALSO **port_set_deallocate()**, **port_set_add()**

port_set_backlog()

SUMMARY Set the size of the port queue

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_set_backlog**(task_t *task*, port_name_t *port_name*, int *backlog*)

ARGUMENTS *task*: The task that has receive rights for the named port (for example, use **task_self()** to specify the caller's task).

port_name: *task*'s name for the port.

backlog: The new backlog to be set.

DESCRIPTION The function **port_set_backlog()** changes the backlog value on the specified port (the port's backlog value is the number of unreceived messages that are allowed in its message queue before the kernel will refuse to accept any more sends to that port).

The task specified by *task* must have receive rights for the named port.

The maximum backlog value is the constant PORT_BACKLOG_MAX. You can get a port's current backlog value by calling **port_status()**.

EXAMPLE

```
#define MY_BACKLOG  10

kern_return_t   error;
port_t          my_port;

error=port_allocate(task_self(),&my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}

error=port_set_backlog(task_self(), my_port, MY_BACKLOG);
if (error!=KERN_SUCCESS)
    mach_error("Call to port_set_backlog failed", error);
```

RETURN KERN_SUCCESS: The backlog value has been changed.

KERN_NOT_RECEIVER: *task* doesn't have receive rights for the port.

KERN_INVALID_ARGUMENT: *task* was invalid, or *port_name* doesn't name a valid port, or the desired backlog wasn't greater than 0, or the desired backlog was greater than PORT_BACKLOG_MAX.

SEE ALSO **msg_send()**, **port_status()**

---

port_set_backup()

SUMMARY Set the backup port for a port

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_set_backup**(task_t *task*, port_name_t *port_name*, port_t *backup*, port_t **previous*)

ARGUMENTS *task*: The task that has receive rights for the named port (for example, use **task_self()** to specify the caller's task).

port_name: *task*'s name for the port right.

backup: The new backup port. If you want to disable the current backup port without setting a new one, set this to PORT_NULL.

previous: Returns the previous backup port.

DESCRIPTION Use this function to keep a port alive despite its being deallocated by its receiver. If the call to **port_set_backup()** is successful, then whenever *port_name* is deallocated by its receiver, *backup* will receive a notification message with receive and send rights for *port_name*. As far as *task* is concerned, the port will be deleted; however, as far as senders to the port are concerned, the port will continue to exist.

To let a port die naturally after its backup port has been set, call **port_set_backup()** on it with *backup* set to PORT_NULL.

EXAMPLE

```
kern_return_t    error;
port_t           my_port, backup_port, previous_port;

error=port_allocate(task_self(),&my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}

error=port_allocate(task_self(),&backup_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}

error=port_set_backup(task_self(), my_port, backup_port,
    &previous_port);
if (error!=KERN_SUCCESS)
    mach_error("Call to port_set_backlog failed", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid, or *port_name* doesn't name a valid port.

KERN_NOT_RECEIVER: *task* doesn't have receive rights for *port_name*.

---

port_set_deallocate()

SUMMARY Destroy a port set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_set_deallocate**(task_t *task*, port_set_name_t *set_name*)

ARGUMENTS *task*: The task that has receive rights for the port set to be destroyed.

set_name: *task*'s name for the doomed port set.

DESCRIPTION The function **port_set_deallocate()** requests that the port set of *task* be destroyed. If the port set isn't empty, any members are first removed.

EXAMPLE

```
kern_return_t   error;
port_set_name_t set_name;

error=port_set_deallocate(task_self(),set_name);
if (error != KERN_SUCCESS) {
    mach_error("port_set_deallocate failed", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid or *set_name* doesn't name a valid port set.

SEE ALSO **port_set_allocate()**

---

port_set_remove()

SUMMARY Remove the named port from a port set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_set_remove**(task_t *task*, port_name_t *port_name*)

ARGUMENTS *task*: The task that has receive rights for the port and port set.

port_name: *task*'s name for the receive rights to be removed.

DESCRIPTION The function **port_set_remove()** removes the named port from a port set. The *task* must have receive rights for the port, and the port must be a member of a port set.

EXAMPLE

```
error=port_set_remove(task_self(), my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_set_remove failed", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_NOT_RECEIVER: *task* doesn't have receive rights for the port.

KERN_NOT_IN_SET: The port isn't a member of a set.

KERN_INVALID_ARGUMENT: *task* was invalid or *port_name* doesn't name a valid port.

SEE ALSO **port_set_add()**

---

port_set_status()

SUMMARY Get the members of a port set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_set_status**(task_t *task*, port_set_name_t *set_name*, port_name_array_t **members*, unsigned int **members_count*)

ARGUMENTS *task*: The task whose port set is queried.

set_name: *task's* name for the port set.

members: Returns *task*'s names for the members of its port set.

members_count: Returns the number of port names in *members*.

DESCRIPTION The function **port_set_status()** returns a list of the ports in a port set. The *members* argument is an array that's automatically allocated when the reply message is received. You should use **vm_deallocate()** on it when the data is no longer needed.

EXAMPLE

```
error=port_set_status(task_self(), set_name, &members,
   &members_count);
if (error != KERN_SUCCESS) {
    mach_error("port_set_status failed", error);
    exit(1);
}
```

```
/* . . . */
error=vm_deallocate(task_self(), (vm_address_t)members,
    sizeof(members)*members_count);
if (error != KERN_SUCCESS)
    mach_error("Trouble freeing members", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid or *set_name* doesn't name a valid port set.

SEE ALSO **port_status()**

---

port_status()

SUMMARY Examine a port's current status

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_status**(task_t *task*, port_name_t *port_name*,
port_set_name_t **port_set_name*, int **num_msgs*, int **backlog*, boolean_t **owner*,
boolean_t **receiver*)

ARGUMENTS *task*: The task that has receive rights for the port in question (for example, use **task_self()** to specify the caller's task).

port_name: *task*'s name for the port right.

port_set_name: Returns *task*'s name for the port set that the named port belongs to, or PORT_NULL if it isn't in a set.

num_msgs: Returns the number of messages queued on this port. If *task* isn't the port's receiver, the number of messages will be returned as negative.

backlog: Returns the number of messages that can be queued to this port without causing the sender to block.

owner: Returns the same value as *receiver*, since ownership rights and receive rights aren't separable.

receiver: Returns true if *task* has receive rights to *port_name*; otherwise, returns false.

DESCRIPTION The function **port_status()** returns the current port status associated with *port_name*.

EXAMPLE

```
error=port_status(task_self(), my_port, &port_set_name, &num_msgs,
    &backlog, &owner, &receiver);
if (error!=KERN_SUCCESS)
    mach_error("Call to port_status failed", error);
```

RETURN KERN_SUCCESS: The data has been retrieved.

KERN_INVALID_ARGUMENT: *task* was invalid or *port_name* doesn't name a valid port.

SEE ALSO **port_set_backlog()**, **port_set_status()**

---

port_type()

SUMMARY Determine the access rights of a task for a specific port name

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **port_type**(task_t *task*, port_name_t *port_name*, port_type_t **port_type*)

ARGUMENTS *task*: The task whose port name space is queried.

port_name: The name being queried.

port_type: Returns a value that indicates what kind of rights the task holds for the port, or whether the name refers to a port set. This value is one of the following: PORT_TYPE_SEND (send rights only), PORT_TYPE_RECEIVE_OWN (send and receive rights), PORT_TYPE_SET (the port is a port set).

DESCRIPTION The function **port_type()** returns information about *task*'s rights for a specific name in its port name space.

EXAMPLE

```
error=port_type(task_self(), port, &type);
if (error != KERN_SUCCESS)
    mach_error("Couldn't get type of port", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid or *task* didn't have any rights named *port_name*.

SEE ALSO **port_names()**, **port_status()**, **port_set_status()**

---

processor_assign(), processor_control(), processor_exit(), processor_get_assignment(), processor_start()

SUMMARY Control a processor

SYNOPSIS

#import <mach/mach.h>

kern_return_t **processor_assign**(processor_t *processor*,
    processor_set_t *new_processor_set*, boolean_t *wait*)
kern_return_t **processor_control**(processor_t *processor*, processor_info_t *info*,
    long **count*)
kern_return_t **processor_exit**(processor_t *processor*)
kern_return_t **processor_get_assignment**(processor_t *processor*,
    processor_set_t **processor_set*)
kern_return_t **processor_start**(processor_t *processor*)

DESCRIPTION **processor_assign()** changes the processor set to which *processor* is assigned. **processor_control()** returns information about *processor*. **processor_exit()** shuts down *processor*. **processor_get_assignment()** returns the processor set to which *processor* is assigned. **processor_start()** starts up *processor*.

Note: These functions are useful only on multiprocessor systems.

processor_info()

SUMMARY Get information about a processor

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_info**(processor_t *processor*, int *flavor*, host_t **host*, processor_info_t *processor_info*, unsigned int **processor_info_count*)

ARGUMENTS *processor*: The processor for which information is to be obtained.

flavor: The type of information that is wanted. Currently only PROCESSOR_BASIC_INFO is implemented.

host: Returns the non-privileged host port for the host on which the processor resides.

processor_info: Returns information about the processor specified by *processor*.

processor_info_count: Size of the info structure. Should be PROCESSOR_BASIC_INFO_COUNT when *flavor* is PROCESSOR_BASIC_INFO.

DESCRIPTION Returns the selected information array for a processor, as specified by *flavor*. The *processor_info* argument is an array of integers that is supplied by the caller and filled with specified information. The *processor_info_count* argument is supplied as the maximum number of integers in *processor_info*. On return, it contains the actual number of integers in *processor_info*.

Basic information is defined by PROCESSOR_BASIC_INFO. The size of this information is defined by PROCESSOR_BASIC_INFO_COUNT. The data structures used by PROCESSOR_BASIC_INFO are defined in the header file **mach/processor_info.h**. Possible values of the **cpu_type** and **cpu_subtype** fields are defined in the header file **mach/machine.h**.

```
typedef int *processor_info_t;  /* variable length array of int */

/* one interpretation of info is */
struct processor_basic_info {
    cpu_type_t      cpu_type;     /* cpu type */
    cpu_subtype_t   cpu_subtype;  /* cpu subtype */
    boolean_t       running;      /* is processor running? */
    int             slot_num;     /* slot number */
    boolean_t       is_master;    /* is this the master processor */
};

typedef struct processor_basic_info *processor_basic_info_t;
```

EXAMPLE

```
kern_return_t                   error;
host_t                          host;
unsigned int                    list_size, info_count;
struct processor_basic_info     info;
processor_array_t               list;

/* Get the processor port. */
error=host_processors(host_priv_self(), &list, &list_size);
if ((error!=KERN_SUCCESS) || (list_size < 1)){
    mach_error("Error calling host_processors (are you root?)",
        error);
    exit(1);
}

/* Get information about the processor. */
info_count=PROCESSOR_BASIC_INFO_COUNT;
error=processor_info(list[0], PROCESSOR_BASIC_INFO, &host,
    (processor_info_t)&info, &info_count);
if (error != KERN_SUCCESS)
    mach_error("Error calling processor_info", error);

/* Now that we're done with the processor port, free it. */
vm_deallocate(task_self(), (vm_address_t)list,
    sizeof(list)*list_size);
if (error!=KERN_SUCCESS)
    mach_error("Trouble freeing list", error);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *processor* isn't a known processor.

MIG_ARRAY_TOO_LARGE: Returned info array is too large for *processor_info*. The *processor_info* argument is filled as much as possible, and *processor_info_count* is set to the number of elements that would be returned if there were enough room.

KERN_FAILURE: *flavor* isn't recognized or *processor_info_count* is too small.

SEE ALSO **processor_start(), processor_exit(), processor_control(), host_processors()**

processor_set_create()

SUMMARY Create a new processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_set_create**(host_t *host*, port_t **new_set*, port_t **new_name*)

DESCRIPTION This function creates a new processor set on *host*.

Note: This function is useful only on multiprocessor systems.

processor_set_default()

SUMMARY Get the port of the default processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_set_default**(host_t *host*, processor_set_t **default_set*)

ARGUMENTS *host*: The host whose default processor set is requested.

default_set: Returns the name (nonprivileged) port for the default processor set.

DESCRIPTION The default processor set is used by all threads, tasks, and processors that aren't explicitly assigned to other sets. This function returns a port that can be used to obtain information about this set (for example, how many threads are assigned to it). This port isn't privileged and thus can't be used to perform operations on that set; call **host_processor_set_priv()** after **processor_set_default()** to get the privileged port.

EXAMPLE

```
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS){
    mach_error("Error calling processor_set_default", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host* is not a host.

SEE ALSO **processor_set_info(), task_assign(), thread_assign()**

---

processor_set_destroy()

SUMMARY Delete a processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_set_destroy**(processor_set_t *processor_set*)

DESCRIPTION This function destroys *processor_set*, reassigning all of its tasks, threads, and processors to the default processor set.

Note: This function is useful only on multiprocessor systems.

---

processor_set_info()

SUMMARY Get information about a processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_set_info**(processor_set_t *processor_set*, int *flavor*, host_t **host*, processor_set_info_t *processor_set_info*, unsigned int **processor_set_info_count*)

ARGUMENTS *processor_set*: The processor set for which information is to be obtained.

flavor: The type of information that is wanted. Should be PROCESSOR_SET_BASIC_INFO or PROCESSOR_SET_SCHED_INFO.

host: Returns the nonprivileged host port for the host on which the processor set resides.

processor_set_info: Returns information about the processor set specified by *processor_set*.

processor_set_info_count: Size of the info structure. Should be PROCESSOR_SET_BASIC_INFO_COUNT when *flavor* is PROCESSOR_SET_BASIC_INFO, and PROCESSOR_SET_SCHED_INFO_COUNT when *flavor* is PROCESSOR_SET_SCHED_INFO.

DESCRIPTION Returns the selected information array for a processor set, as specified by *flavor*. The *processor_set_info* argument is an array of integers that is supplied by the caller, and filled with specified information. The *processor_set_info_count* argument is supplied as the maximum number of integers in *processor_set_info*. On return, it contains the actual number of integers in *processor_set_info*.

Basic information is defined by PROCESSOR_SET_BASIC_INFO. The size of this information is defined by PROCESSOR_SET_BASIC_INFO_COUNT. The **load_average** and **mach_factor** fields are scaled by the constant LOAD_SCALE (that is, the integer value returned is the load average or Mach factor multiplied by LOAD_SCALE).

The *Mach factor*, like the UNIX load average, is a measurement of how busy the system is. Unlike the load average, higher Mach factors mean that the system is less busy. The Mach factor tells you how much of a CPU you have available for running an application. For example, on a single-processor system with one job running, the Mach factor is 0.5; this means if another job starts running it will get half of the CPU. (Two jobs will be running, each getting half the CPU.) On a single-processor system, the Mach factor is between zero and one. On a multiprocessor system, the Mach factor can go over one. For example, a three-processor system with one job running has a Mach factor of 2.0, since two processors are available to new jobs.

```
struct processor_set_basic_info {
    int  processor_count;    /* number of processors */
    int  task_count;         /* number of tasks */
    int  thread_count;       /* number of threads */
    int  load_average;       /* scaled load average */
    int  mach_factor;        /* scaled mach factor */
};
typedef struct processor_set_basic_info *processor_set_basic_info_t;
```

Scheduling information is defined by PROCESSOR_SET_SCHED_INFO. The size of this information is given by PROCESSOR_SET_SCHED_INFO_COUNT.

```
struct processor_set_sched_info {
    int  policies;          /* allowed policies */
    int  max_priority;      /* max priority for new threads */
};
typedef struct processor_set_sched_info *processor_set_sched_info_t;
```

EXAMPLE

```
kern_return_t                    error;
host_t                           host;
unsigned int                     info_count;
struct processor_set_basic_info  info;
processor_set_t                  default_set;

error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS){
    mach_error("Error calling processor_set_default", error);
    exit(1);
}

info_count=PROCESSOR_SET_BASIC_INFO_COUNT;
error=processor_set_info(default_set, PROCESSOR_SET_BASIC_INFO,
   &host, (processor_set_info_t)&info, &info_count);
if (error != KERN_SUCCESS)
    mach_error("Error calling processor_set_info", error);

printf("The UNIX load average is %f\n",
    (float)info.load_average/LOAD_SCALE);
printf("The Mach factor is %f\n", (float)info.mach_factor/LOAD_SCALE);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *processor_set* is not a processor set, or *flavor* is not recognized.

KERN_FAILURE: *processor_set_info_count* is less than what it should be.

MIG_ARRAY_TOO_LARGE: Returned info array is too large for *processor_set_info*.

SEE ALSO

processor_set_create(), processor_set_default(), processor_assign(), task_assign(), thread_assign()

processor_set_max_priority()

SUMMARY Set the maximum priority permitted on a processor set

SYNOPSIS

#import <mach/mach.h>

kern_return_t **processor_set_max_priority**(processor_set_t *processor_set*, int *max_priority*, boolean_t *change_threads*)

DESCRIPTION This function affects only newly created or newly assigned threads unless you specify *change_threads* as true.

Note: This function is useful only on multiprocessor systems.

processor_set_policy_enable(), processor_set_policy_disable()

SUMMARY Enable or disable a scheduling policy on a processor set

SYNOPSIS

#import <mach/mach.h>

kern_return_t **processor_set_policy_enable**(processor_set_t *processor_set*, int *policy*)
kern_return_t **processor_set_policy_disable**(processor_set_t *processor_set*, int *policy*, boolean_t *change_threads*)

ARGUMENTS *processor_set*: The processor set whose allowed policies are to be changed. This must be the privileged processor set port, which is returned by **host_processor_set_priv**().

policy: The policy to enable or disable. Currently, the only valid policies are POLICY_TIMESHARE, POLICY_INTERACTIVE, and POLICY_FIXEDPRI. You can't disable timesharing.

change_threads: Specify true if you want to reset to timesharing the policies of any threads with the newly disallowed policy. Otherwise, specify false.

DESCRIPTION Processor sets may restrict the scheduling policies to be used for threads assigned to them. These two calls provide the mechanism for designating permitted and forbidden policies. The current set of permitted policies can be obtained from **processor_set_info()**. Timesharing can't be forbidden by any processor set. This is a compromise to reduce the complexity of the assign operation; any thread whose policy is forbidden by the target processor set has its policy reset to timesharing. If the *change_threads* argument to **processor_set_policy_disable()** is true, threads currently assigned to this processor set and using the newly disabled policy will have their policy reset to timesharing.

Warning: Don't use POLICY_FIXEDPRI unless you're familiar with the consequences of fixed-priority scheduling. Using fixed-priority scheduling in a process can keep other processes from getting any CPU time. If processes that are essential to the functioning of the system don't get CPU time, you might have to reboot your system to make it work normally.

EXAMPLE

```
kern_return_t    error;
processor_set_t  default_set, default_set_priv;

error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error != KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

error=processor_set_policy_enable(default_set_priv, POLICY_FIXEDPRI);
if (error != KERN_SUCCESS)
    mach_error("Error calling processor_set_policy_enable", error);
```

RETURN KERN_SUCCESS: Operation completed successfully.

KERN_INVALID_ARGUMENT: *processor_set* isn't the privileged port of a processor set, *policy* isn't a valid policy, or an attempt was made to disable timesharing.

SEE ALSO **thread_policy(), thread_switch()**

processor_set_tasks()

SUMMARY Get kernel ports for tasks assigned to a processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_set_tasks**(processor_set_t *processor_set*,
task_array_t **task_list*, unsigned int **task_count*)

ARGUMENTS *processor_set*: The processor set to be affected. This must be the privileged processor set port, which is returned by **host_processor_set_priv()**.

task_list: Returns the set of tasks currently assigned to *processor_set*; no particular ordering is guaranteed.

task_count: Returns the number of tasks in *task_list*.

DESCRIPTION This function gets send rights to the kernel port for each task currently assigned to *processor_set*. The *task_list* argument is an array that is created as a result of this call. You should call **vm_deallocate()** on this array when you no longer need the data.

EXAMPLE

```
task_array_t    task_list;
unsigned int    task_count;
processor_set_t default_set, default_set_priv;
kern_return_t   error;

error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error != KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}
```

```
error=processor_set_tasks(default_set_priv, &task_list, &task_count);
if (error != KERN_SUCCESS) {
    mach_error("Call to processor_set_tasks() failed", error);
    exit(1);
}

/* . . . */
error=vm_deallocate(task_self(), (vm_address_t)task_list,
    sizeof(task_list)*task_count);
if (error != KERN_SUCCESS)
    mach_error("Trouble freeing task_list", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *processor_set* isn't a privileged processor set.

SEE ALSO **task_assign()**, **thread_assign()**, **processor_set_threads()**

---

processor_set_threads()

SUMMARY Get kernel ports for threads assigned to a processor set

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **processor_set_threads**(processor_set_t *processor_set*,
thread_array_t **thread_list*, unsigned int **thread_count*)

ARGUMENTS *processor_set*: The processor set to be affected. This must be the privileged processor set port, which is returned by **host_processor_set_priv()**.

thread_list: Returns the set of threads currently assigned to *processor_set*; no particular ordering is guaranteed.

thread_count: Returns the number of threads in *thread_list*.

DESCRIPTION This function gets send rights to the kernel port for each thread currently assigned to *processor_set*. The *thread_list* argument is an array that is created as a result of this call. You should call **vm_deallocate()** on *thread_list* when you no longer need the data.

EXAMPLE

```
thread_array_t  thread_list;
unsigned int    thread_count;
processor_set_t default_set, default_set_priv;
kern_return_t   error;

error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error != KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

error=processor_set_threads(default_set_priv, &thread_list, &thread_count);
if (error != KERN_SUCCESS) {
    mach_error("Call to processor_set_threads() failed", error);
    exit(1);
}

/* . . . */
error=vm_deallocate(task_self(), (vm_address_t)thread_list,
    sizeof(thread_list)*thread_count);
if (error != KERN_SUCCESS)
    mach_error("Trouble freeing thread_list", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *processor_set* isn't a privileged processor set.

SEE ALSO **task_assign(), thread_assign(), processor_set_tasks()**

processor_start() → See processor_assign()

task_assign(), task_assign_default()

SUMMARY Assign a task to a processor set

SYNOPSIS

#import <mach/mach.h>

kern_return_t **task_assign**(task_t *task*, processor_set_t *new_processor_set*, boolean_t *assign_threads*)

kern_return_t **task_assign_default**(task_t *task*, boolean_t *assign_threads*)

DESCRIPTION The **task_assign()** function assigns *task* to *new_processor_set*; **task_assign_default()** assigns *task* to the default processor set.

Note: These functions are useful only on multiprocessor systems.

task_by_unix_pid()

SUMMARY Get the task port for a UNIX process on the same host

SYNOPSIS

#import <mach/mach.h>

kern_return_t **task_by_unix_pid**(task_t *task*, int *pid*, task_t **result_task*)

ARGUMENTS *task*: A task that is used to check permission (usually **task_self()**).

pid: The process ID of the desired process.

result_task: Returns send rights to the task port of the process specified by *pid*.

DESCRIPTION Returns the task port for another process, named by its process ID, on the same host as *task*. This call succeeds only if the caller is the superuser or *task* has the same user ID as the process specified by *pid*. If the call fails, *result_task* is set to TASK_NULL.

EXAMPLE

```
pid=fork();

if (pid==0) /* We're in the child. */ {
    /* do childish things */
}
else /* We're in the parent */ {
    result=task_by_unix_pid(task_self(), pid, &child_task);
    if (result != KERN_SUCCESS)
        mach_error("task_by_unix_pid", result);
    /* . . . */
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_FAILURE: *target_task* has a different user ID from the process corresponding to *pid,* and the caller isn't the superuser; or *pid* didn't refer to a valid process; or *target_task* wasn't a valid task.

SEE ALSO **unix_pid()**

---

task_create()

SUMMARY Create a task

SYNOPSIS

#import <mach/mach.h>

kern_return_t **task_create**(task_t *parent_task*, boolean_t *inherit_memory*, task_t **child_task*)

ARGUMENTS

parent_task: The task from which the child's capabilities are drawn.

inherit_memory: If set, the address space of the child task is built from the parent task according to its memory inheritance values; otherwise, the child task is given an empty address space.

child_task: Returns the new task.

DESCRIPTION The function **task_create()** creates a new task from *parent_task*; the resulting task (*child_task*) acquires shared or copied parts of the parent's address space (see **vm_inherit()**). The child task initially has no threads; you put threads in it using **thread_create()**.

Important: Normally, you should use the UNIX **fork()** system call instead of **task_create()**.

The child task gets the four special ports initialized for it at task creation. The kernel port (task port) is created, and send rights for it are given to the child and returned to the caller in *child_task*. The notify port is initialized to null. The child inherits its bootstrap and exception ports from the parent task. The new task can get send rights to these ports with the call **task_get_special_port()**.

EXAMPLE

```
error=task_create(task_self(), TRUE, &child_task);
if(error!=KERN_SUCCESS)
    mach_error("Call to task_create() failed", error);
```

RETURN KERN_SUCCESS: A new task has been created.

KERN_INVALID_ARGUMENT: *parent_task* is not a valid task port.

KERN_RESOURCE_SHORTAGE: Some critical kernel resource is unavailable.

SEE ALSO **task_terminate()**, **task_suspend()**, **task_resume()**, **task_get_special_port()**, **task_set_special_port()**, **task_self()**, **task_threads()**, **thread_create()**, **thread_resume()**, **vm_inherit()**

---

task_get_assignment()

SUMMARY Get the name of the processor set that a task is assigned to

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_get_assignment**(task_t *task*, processor_set_t **processor_set*)

Note: This function is useful only on multiprocessor systems.

task_get_special_port(), task_set_special_port(), task_self(), task_notify()

SUMMARY Get or set a task's special ports

SYNOPSIS

#import <mach/mach.h>

kern_return_t **task_get_special_port**(task_t *task*, int *which_port*, port_t **special_port*)
kern_return_t **task_set_special_port**(task_t *task*, int *which_port*, port_t *special_port*)
task_t **task_self**(void)
port_t **task_notify**(void)

ARGUMENTS *task*: The task to get the port for.

which_port: The port that's requested. This is one of:

TASK_NOTIFY_PORT
TASK_BOOTSTRAP_PORT
TASK_EXCEPTION_PORT

special_port: The value of the port that's being requested or set.

DESCRIPTION The function **task_get_special_port()** returns send rights to one of a set of special ports for the task specified by *task*. In the case of the task's own notify port, the task also gets receive rights.

The function **task_set_special_port()** sets one of a set of special ports for the task specified by *task*.

The function **task_self()** returns the port to which kernel calls for the currently executing thread should be directed. Currently, **task_self()** returns the task kernel port, which is a port for which the kernel has receive rights and which it uses to identify a task. In the future it may be possible for one task to interpose a port as another task's kernel port. At that time **task_self()** will still return the port to which the executing thread should direct kernel calls, but it may no longer be a port for which the kernel has receive rights.

If a controller task has send access to the kernel port of a subject task, then the controller task can perform kernel operations for the subject task. Normally, only the task itself and the task that created it will have access to the task kernel port, but any task may pass rights to its kernel port to any other task.

The function **task_notify()** returns receive and send rights to the notify port associated with the task to which the executing thread belongs. The notify port is a port on which the task should receive notification of such kernel events as the destruction of a port to which it has send rights.

The other special ports associated with a task are the bootstrap port and the exception port. The bootstrap port is a port to which a thread may send a message requesting other system service ports. This port isn't used by the kernel. The task's exception port is the port to which messages are sent by the kernel when an exception occurs and the thread causing the exception has no exception port of its own.

Important: If you set your task's bootstrap port, you should also set the global variable **bootstrap_port** to *special_port*. The **bootstrap_port** variable is task-wide and is used by **mach_init** and other processes to determine your task's bootstrap port. Since you can't change the value of the **bootstrap_port** variable in another task, you should use care when changing the bootstrap port of another task.

MACRO EQUIVALENTS

The following macros are defined in the header file **mach/task_special_ports.h**:

task_get_notify_port(*task*, *port*)
task_set_notify_port(*task*, *port*)

task_get_exception_port(*task*, *port*)
task_set_exception_port(*task*, *port*)

task_get_bootstrap_port(*task*, *port*)
task_set_bootstrap_port(*task*, *port*)

EXAMPLE

```
/* Save the old exception port for this task. */
r = task_get_exception_port(task_self(), &(ports.old_exc_port));
if (r != KERN_SUCCESS) {
    mach_error("task_get_exception_port", r);
    exit(1);
}

    /* Create a new exception port for this task. */
r = port_allocate(task_self(), &(ports.exc_port));
if (r != KERN_SUCCESS) {
    mach_error("port_allocate 0", r);
    exit(1);
}
r = task_set_exception_port(task_self(), (ports.exc_port));
if (r != KERN_SUCCESS) {
    mach_error("task_set_exception_port", r);
    exit(1);
}
```

RETURN KERN_SUCCESS: The port was returned or set.

KERN_INVALID_ARGUMENT: Either *task* is not a task or *which_port* is an invalid port selector.

SEE ALSO **thread_special_ports(), task_create()**

---

task_info()

SUMMARY Get information about a task

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_info**(task_t *target_task*, int *flavor*, task_info_t *task_info*, unsigned int **task_info_count*)

ARGUMENTS *target_task*: The task to be affected (for example, use **task_self()** to specify the caller's task).

flavor: The type of statistics that are wanted. Currently only TASK_BASIC_INFO is implemented.

task_info: Returns statistics about *target_task*.

task_info_count: Size of the info structure. Currently this must be TASK_BASIC_INFO_COUNT.

DESCRIPTION The function **task_info()** returns the information specified by *flavor* about a task. The *task_info* argument is an array of integers that's supplied by the caller and returned filled with information. The *task_info_count* argument is supplied as the maximum number of integers in *task_info*. On return, it contains the actual number of integers in *task_info*.

Currently there's only one flavor of information, defined by TASK_BASIC_INFO. Its size is defined by TASK_BASIC_INFO_COUNT. The definition of the information structure returned by TASK_BASIC_INFO is:

```
struct task_basic_info {
    int             suspend_count;  /* suspend count for task */
    int             base_priority;  /* base scheduling priority */
    vm_size_t       virtual_size;   /* number of virtual pages */
    vm_size_t       resident_size;  /* number of resident pages */
    time_value_t    user_time;      /* total user run time for
                                       terminated threads */
    time_value_t    system_time;    /* total system run time for
                                       terminated threads */
};
typedef struct task_basic_info  *task_basic_info_t;
```

EXAMPLE

```
kern_return_t               error;
struct task_basic_info      info;
unsigned int                info_count=TASK_BASIC_INFO_COUNT;

error=task_info(task_self(), TASK_BASIC_INFO,
    (task_info_t)&info, &info_count);
if (error!=KERN_SUCCESS)
    mach_error("Error calling task_info()", error);
else
    printf("Base priority is %d\n", info.base_priority);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *target_task* isn't a task, or *flavor* isn't recognized.

MIG_ARRAY_TOO_LARGE: The returned info array is too large for *task_info*. The *task_info* argument is filled as much as possible, and *task_info_count* is set to the number of elements that would be returned if there were enough room.

SEE ALSO **task_threads(), thread_info(), thread_get_state()**

task_notify() → See task_get_special_port()

---

task_priority()

SUMMARY Set the scheduling priority for a task

SYNOPSIS

#import <mach/mach.h>

kern_return_t **task_priority**(task_t *task*, int *priority*, boolean_t *change_threads*)

ARGUMENTS *task*: Task to set priority for.

priority: New priority.

change_threads: Change priority of existing threads if true.

DESCRIPTION The priority of a task is used only for creation of new threads; the priority of a new thread priority is set to that of its task. The **task_priority()** function changes this task priority. It also sets the priorities of all threads in the task to this new priority if *change_threads* is true. Existing threads are not affected otherwise. If this priority change violates the maximum priority of some threads, as many threads as possible will be changed and an error code will be returned.

Priorities range from 0 to 31, where higher numbers denote higher priorities. You can retrieve the current scheduling priority using **thread_info()**.

EXAMPLE

```
kern_return_t              error;
struct task_basic_info     info;
unsigned int               info_count=TASK_BASIC_INFO_COUNT;

error=task_info(task_self(), TASK_BASIC_INFO,
    (task_info_t)&info, &info_count);
if (error!=KERN_SUCCESS)
    mach_error("Error calling task_info()", error);
else {
    /* Set this task's base priority to be much lower than normal */
    error = task_priority(task_self(), info.base_priority - 4, TRUE);
    if (error != KERN_SUCCESS)
        mach_error("Call to task_priority() failed", error);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* is not a task, or *priority* is not a valid priority.

KERN_FAILURE: *change_threads* was true and the attempt to change the priority of at least one existing thread failed because the new priority would have exceeded that thread's maximum priority.

SEE ALSO **thread_priority()**, **processor_set_max_priority()**, **thread_switch()**

---

task_resume()

SUMMARY Resume a task

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_resume**(task_t *target_task*)

ARGUMENTS *target_task*: The task to be resumed.

DESCRIPTION The function **task_resume()** decrements the task's suspend count. If the suspend count becomes 0, all threads with 0 suspend counts in the task are resumed. If the suspend count is already 0, it's not decremented (it never becomes negative).

RETURN KERN_SUCCESS: The task has been resumed.

KERN_FAILURE: The suspend count is already 0.

KERN_INVALID_ARGUMENT: *target_task* isn't a task.

SEE ALSO **task_create()**, **task_terminate()**, **task_suspend()**, **task_info()**, **thread_suspend()**, **thread_resume()**, **thread_info()**

task_self() → See task_get_special_port()

task_set_special_port() → See task_get_special_port()

task_suspend()

SUMMARY Suspend a task

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_suspend**(task_t *target_task*)

ARGUMENTS *target_task*: The task to be suspended (for example, use **task_self()** to specify the caller's task).

DESCRIPTION The function **task_suspend()** increments the task's suspend count and stops all threads in the task. As long as the suspend count is positive, newly created threads will not run. This call doesn't return until all threads are suspended.

If the count becomes greater than 1, it will take more than one **task_resume()** call to restart the task.

RETURN KERN_SUCCESS: The task has been suspended.

KERN_INVALID_ARGUMENT: *target_task* isn't a task.

SEE ALSO **task_create()**, **task_terminate()**, **task_resume()**, **task_info()**, **thread_suspend()**

---

task_terminate()

SUMMARY Terminate a task

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_terminate**(task_t *target_task*)

ARGUMENTS *target_task*: The task to be destroyed (for example, use **task_self()** to specify the caller's task).

DESCRIPTION The function **task_terminate()** destroys the task specified by *target_task* and all its threads. All resources that are used only by this task are freed. Any port to which this task has receive rights is destroyed.

RETURN KERN_SUCCESS: The task has been destroyed.

KERN_INVALID_ARGUMENT: *target_task* isn't a task.

SEE ALSO **task_create()**, **task_suspend()**, **task_resume()**, **thread_terminate()**, **thread_suspend()**

---

task_threads()

SUMMARY Get a task's threads

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **task_threads**(task_t *target_task*, thread_array_t **thread_list*, unsigned int **thread_count*)

ARGUMENTS *target_task*: The task to be affected (for example, use **task_self()** to specify the caller's task).

thread_list: Returns the set of threads contained within *target_task*; no particular ordering is guaranteed.

thread_count: Returns the number of threads in *thread_list*.

DESCRIPTION The function **task_threads()** gets send rights to the kernel port for each thread contained in *target_task*.

The array *thread_list* is created as a result of this call. You should call **vm_deallocate()** on this array when the data is no longer needed.

EXAMPLE

```
r = task_threads(task_self(), &thread_list, &thread_count);
if (r != KERN_SUCCESS)
    mach_error("Error calling task_threads", r);
else {
    if (thread_count == 1)
        printf ("There's 1 thread in this task\n");
    else
        printf("There are %d threads in this task\n", thread_count);

/* Deallocate the list of threads. */
    r = vm_deallocate(task_self(), (vm_address_t)thread_list,
        sizeof(thread_list)*thread_count);
    if (r != KERN_SUCCESS)
        mach_error("Trouble freeing thread_list", r);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *target_task* isn't a task.

SEE ALSO **thread_create(), thread_terminate(), thread_suspend()**

---

thread_abort()

SUMMARY Interrupt a thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_abort**(thread_t *target_thread*)

ARGUMENTS *target_thread*: The thread to be interrupted.

DESCRIPTION The function **thread_abort()** aborts page faults and the kernel functions **msg_send()**, **msg_receive()**, and **msg_rpc()**, making the call return a code indicating that it was interrupted. The call is interrupted whether or not the thread (or task containing it) is currently suspended. If it's suspended, the thread receives the interrupt when it resumes.

A thread will retry an aborted page fault if its state isn't modified before it resumes. The function **msg_send()** returns SEND_INTERRUPTED; **msg_receive()** returns RCV_INTERRUPTED; and **msg_rpc()** returns either SEND_INTERRUPTED or RCV_INTERRUPTED, depending on which half of the RPC was interrupted.

This function lets one thread stop another thread cleanly, thereby allowing the future execution of the target thread to be controlled in a predictable way. The **thread_suspend()** function keeps the target thread from executing any further instructions at the user level, including the return from a system call. The **thread_get_state()** and **thread_set_state()** functions let you examine or modify the user state of a target thread. However, if a suspended thread was executing within a system call, it also has associated with it a kernel state. This kernel state can't be modified by **thread_set_state()**; therefore, when the thread is resumed the system call may return, changing the user state and possibly user memory.

The **thread_abort()** function aborts the kernel call from the target thread's point of view by resetting the kernel state so that the thread will resume execution just after the system call. The system call will return one of the interrupted codes described previously. The system call will either be entirely completed or entirely aborted, depending on the precise moment at which the abort was received. Thus if the thread's user state has been changed by **thread_set_state()**, it won't be modified by any unexpected system call side effects.

For example, to simulate a UNIX signal, the following sequence of calls may be used:

1. **thread_suspend()**—Stops the thread.
2. **thread_abort()**—Interrupts any system call in progress, setting the return value to "interrupted." Since the thread is stopped, it won't return to user code.
3. **thread_set_state()**—Alters the thread's state to simulate a procedure call to the signal handler.
4. **thread_resume()**—Resumes execution at the signal handler. If the thread's stack has been correctly set up, the thread can return to the interrupted system call.

Calling **thread_abort()** on a thread that's not suspended is risky, since it's difficult to know exactly what system trap, if any, the thread might be executing and whether an interrupt return would cause the thread to do something useful.

RETURN KERN_SUCCESS: The thread received an interrupt.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread.

SEE ALSO **thread_get_state()**, **thread_info()**, **thread_terminate()**, **thread_suspend()**

thread_assign(), thread_assign_default()

SUMMARY Assign a thread to a processor set

SYNOPSIS

#import <mach/mach.h>

kern_return_t **thread_assign**(thread_t *thread*, processor_set_t *new_processor_set*)
kern_return_t **thread_assign_default**(thread_t *thread*)

DESCRIPTION **thread_assign**() assigns *thread* to *new_processor_set*; **thread_assign_default**() assigns *thread* to the default processor set.

Note: These functions are useful only on multiprocessor systems.

thread_create()

SUMMARY Create a thread

SYNOPSIS

#import <mach/mach.h>

kern_return_t **thread_create**(task_t *parent_task*, thread_t **child_thread*)

ARGUMENTS *parent_task*: The task that should contain the new thread.

child_thread: Returns the new thread.

DESCRIPTION The function **thread_create()** creates a new thread within *parent_task*. The new thread has no processor state, and has a suspend count of 1. To get a new thread to run, first call **thread_create()** to get the new thread's identifier, *child_thread*. Then call **thread_set_state()** to set a processor state. Finally, call **thread_resume()** to schedule the thread to execute.

Important: Don't use this function unless you're writing a loadable kernel server or implementing a new thread package, such as the C-thread functions. For normal, user-level programming, use **cthread_fork()** instead. You can then use **cthread_thread()** if you need to get the Mach thread that corresponds to the new C-thread.

When the thread is created, send rights to its thread kernel port are given to it and returned to the caller in *child_thread*. The new thread's exception port is set to PORT_NULL.

RETURN KERN_SUCCESS: A new thread has been created.

KERN_INVALID_ARGUMENT: *parent_task* isn't a valid task.

KERN_RESOURCE_SHORTAGE: Some critical kernel resource isn't available.

SEE ALSO **task_create(), task_threads(), thread_terminate(), thread_suspend(), thread_resume(), thread_special_ports(), thread_set_state()**

---

thread_get_assignment()

SUMMARY Get the name of the processor set to which a thread is assigned

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_get_assignment**(thread_t *thread*, processor_set_t **processor_set*)

Note: This function is useful only in multiprocessor systems.

---

thread_get_special_port(), thread_set_special_port(), thread_self(), thread_reply()

SUMMARY Get or set a thread's special ports

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_get_special_port**(thread_t *thread*, int *which_port*,
port_t **special_port*)
kern_return_t **thread_set_special_port**(thread_t *thread*, int *which_port*,
port_t *special_port*)
thread_t **thread_self**(void)
port_t **thread_reply**(void)

ARGUMENTS *thread*: The thread to get the port for.

which_port: The port that's requested. This is one of:

THREAD_REPLY_PORT
THREAD_EXCEPTION_PORT

special_port: The value of the port that's being requested or set.

DESCRIPTION The function **thread_get_special_port()** returns send rights to one of a set of special ports for the thread specified by *thread.* In the case of getting the thread's own reply port, receive rights are also given to the thread.

The function **thread_set_special_port()** sets one of a set of special ports for the thread specified by *thread.*

The function **thread_self()** returns the port to which kernel calls for the currently executing thread should be directed. Currently **thread_self()** returns the thread kernel port, which is a port for which the kernel has receive rights and which it uses to identify a thread. In the future it may be possible for one thread to interpose a port as another thread's kernel port. At that time **thread_self()** will still return the port to which the executing thread should direct kernel calls, but it may no longer be a port for which the kernel has receive rights.

If a controller thread has send access to the kernel port of a subject thread, the controller thread can perform kernel operations for the subject thread. Normally only the thread itself and its parent task will have access to the thread kernel port, but any thread may pass rights to its kernel port to any other thread.

The function **thread_reply()** returns receive and send rights to the reply port of the calling thread. The reply port is a port to which the thread has receive rights. It's used to receive any initialization messages and as a reply port for early remote procedure calls.

A thread also has access to its task's special ports.

MACRO EQUIVALENTS The following macros are defined in the header file **mach/thread_special_ports.h**:

thread_get_reply_port(*thread, port*)
thread_set_reply_port(*thread, port*)

thread_get_exception_port(*thread, port*)
thread_set_exception_port(*thread, port*)

RETURN KERN_SUCCESS: The port was returned or set.

KERN_INVALID_ARGUMENT: *thread* isn't a thread, or *which_port* is an invalid port selector.

SEE ALSO **task_get_special_port(), task_set_special_port(), task_self(), thread_create()**

---

thread_get_state(), thread_set_state()

SUMMARY Get or set a thread's state

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_get_state**(thread_t *target_thread*, int *flavor*,
thread_state_data_t *old_state*, unsigned int **old_state_count*)
kern_return_t **thread_set_state**(thread_t *target_thread*, int *flavor*,
thread_state_data_t *new_state*, unsigned int *new_state_count*)

ARGUMENTS *target_thread*: The thread whose state is affected.

flavor: The type of state that's to be manipulated. This may be any one of the following:

NeXT_THREAD_STATE_REGS
NeXT_THREAD_STATE_68882
NeXT_THREAD_STATE_USER_REG

old_state: Returns an array of state information.

new_state: An array of state information.

old_state_count: The size of the state information array. This may be any one of the following:

NeXT_THREAD_STATE_REGS_COUNT
NeXT_THREAD_STATE_68882_COUNT
NeXT_THREAD_STATE_USER_REG_COUNT

new_state_count: Same as *old_state_count*.

DESCRIPTION The function **thread_get_state()** returns the state component (that is, the machine registers) of *target_thread* as specified by *flavor*. The *old_state* is an array of integers that's provided by the caller and returned filled with the specified information. You should set *old_state_count* to the maximum number of integers in *old_state*. On return, *old_state_count* is equal to the actual number of integers in *old_state*.

The function **thread_set_state()** sets the state component of *target_thread* as specified by *flavor*. The *new_state* is an array of integers that the caller fills. You should set *new_state_count* to the number of elements in *new_state*. The entire set of registers is reset.

target_thread must not be **thread_self()** for either of these calls.

The state structures are defined in the header file **mach/machine/thread_status.h**.

RETURN KERN_SUCCESS: The state has been set or returned.

MIG_ARRAY_TOO_LARGE: The returned state is too large for the *new_state*. The *new_state* argument is filled in as much as possible, and *new_state_count* is set to the number of elements that would be returned if there were enough room.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread, *target_thread* is **thread_self()**, or *flavor* is unrecognized for this machine.

SEE ALSO **task_info()**, **thread_info()**

---

thread_info()

SUMMARY Get information about a thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_info**(thread_t *target_thread*, int *flavor*, thread_info_t *thread_info*, unsigned int **thread_info_count*)

ARGUMENTS *target_thread*: The thread to be affected.

flavor: The type of statistics wanted. This can be THREAD_BASIC_INFO or THREAD_SCHED_INFO.

thread_info: Returns statistics about *target_thread.*

thread_info_count: Size of the info structure. This can be THREAD_BASIC_INFO_COUNT or THREAD_SCHED_INFO_COUNT.

DESCRIPTION The function **thread_info()** returns the selected information array for a thread, as specified by *flavor.* The *thread_info* argument is an array of integers that's supplied by the caller and returned filled with specified information. The *thread_info_count* argument is supplied as the maximum number of integers in *thread_info.* On return, it contains the actual number of integers in *thread_info.*

The size of the information returned by THREAD_BASIC_INFO is defined by THREAD_BASIC_INFO_COUNT. The definition of the information structure returned by THREAD_BASIC_INFO is:

```
struct thread_basic_info {
    time_value_t  user_time;      /* user run time */
    time_value_t  system_time;    /* system run time */
    int           cpu_usage;      /* scaled cpu usage percentage */
    int           base_priority;  /* base scheduling priority */
    int           cur_priority;   /* current scheduling priority */
    int           run_state;      /* run state */
    int           flags;          /* various flags */
    int           suspend_count;  /* suspend count for thread */
    long          sleep_time;     /* number of seconds that thread
                                     has been sleeping */
};
typedef struct thread_basic_info *thread_basic_info_t;
```

The **run_state** field has one of the following values:

TH_STATE_RUNNING: The thread is running normally.

TH_STATE_STOPPED: The thread is suspended. This happens when the thread or task suspend count is greater than zero.

TH_STATE_WAITING: The thread is sleeping normally.

TH_STATE_UNINTERRUPTIBLE: The thread is in an uninterruptible sleep. This should happen only for very short times during some system calls.

TH_STATE_HALTED: The thread is halted at a clean point. This state happens only after a call to **thread_abort()**.

Possible values of the **flags** field are:

TH_FLAGS_SWAPPED: The thread is swapped out. This happens when the thread hasn't run in a long time, and the kernel stack for the thread has been swapped out.

TH_FLAGS_IDLE: The thread is the idle thread for the CPU. This means that the CPU runs this thread whenever it has no other threads to run.

The **sleep_time** field is useful only when **run_state** is TH_STATE_STOPPED. (Currently **sleep_time** is always set to zero, no matter how long the thread has been sleeping.)

The size of the information returned by THREAD_SCHED_INFO is defined by THREAD_SCHED_INFO_COUNT. The definition of the information structure returned by THREAD_SCHED_INFO is:

```
struct thread_sched_info {
    int       policy;           /* scheduling policy */
    int       data;             /* associated data */
    int       base_priority;    /* base priority */
    int       max_priority;     /* max priority */
    int       cur_priority;     /* current priority */
    boolean_t depressed;        /* depressed ? */
    int       depress_priority; /* priority depressed from */
};
typedef struct thread_sched_info    *thread_sched_info_t;
```

The **policy** field has one of the following values: POLICY_FIXEDPRI, POLICY_TIMESHARE, or POLICY_INTERACTIVE. If **policy** is POLICY_FIXEDPRI, then **data** is the quantum (in milliseconds). Otherwise, **data** is meaningless.

EXAMPLE Example of using THREAD_BASIC_INFO:

```
kern_return_t             error;
struct thread_basic_info  info;
unsigned int              info_count=THREAD_BASIC_INFO_COUNT;

error=thread_info(thread_self(), THREAD_BASIC_INFO,
    (thread_info_t)&info, &info_count);
if (error!=KERN_SUCCESS)
    mach_error("Error calling thread_info()", error);
else {
    printf("User time is %d seconds, %d microseconds\n",
        info.user_time.seconds, info.user_time.microseconds);
    printf("System time is %d seconds, %d microseconds\n",
        info.system_time.seconds, info.system_time.microseconds);
}
```

Example of using THREAD_SCHED_INFO:

```
kern_return_t              error;
struct thread_sched_info   info;
unsigned int               info_count=THREAD_SCHED_INFO_COUNT;

error=thread_info(thread_self(), THREAD_SCHED_INFO,
    (thread_info_t)&info, &info_count);
if (error!=KERN_SUCCESS)
    mach_error("Error calling thread_info()", error);
else {
    printf("Base priority is %d\n", info.base_priority);
    printf("Max priority is %d\n", info.max_priority);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread, or *flavor* isn't recognized, or *thread_info_count* is smaller than it's supposed to be.

MIG_ARRAY_TOO_LARGE: The returned info array is too large for *thread_info*. The *thread_info* argument is filled as much as possible, and *thread_info_count* is set to the number of elements that would have been returned if there were enough room.

SEE ALSO **thread_get_special_port(), task_threads(), task_info(), thread_get_state()**

thread_max_priority() → See thread_priority()

---

thread_policy()

SUMMARY Set scheduling policy for a thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_policy**(thread_t *thread*, int *policy*, int *data*)

ARGUMENTS *thread*: Thread to set policy for.

policy: Policy to set. This must be POLICY_TIMESHARE, POLICY_INTERACTIVE, or POLICY_FIXEDPRI.

data: Policy-specific data.

DESCRIPTION This function changes the scheduling policy for *thread* to *policy*.

The *data* argument is meaningless for the timesharing and interactive policies; for the fixed-priority policy, it's the quantum to be used (in milliseconds). The system will always round the quantum up to the next multiple of the basic system quantum (**min_quantum**, which can be obtained from **host_info()**). You can find the current quantum using **thread_info()**.

Processor sets can restrict the allowed policies, so this call will fail if the processor set to which *thread* is currently assigned doesn't permit *policy*.

EXAMPLE

```
kern_return_t           error;
struct host_sched_info  sched_info;
unsigned int            sched_count=HOST_SCHED_INFO_COUNT;
int                     quantum;
processor_set_t         default_set, default_set_priv;

/* Set quantum to a reasonable value. */
error=host_info(host_self(), HOST_SCHED_INFO,
    (host_info_t)&sched_info, &sched_count);
if (error != KERN_SUCCESS) {
    mach_error("SCHED host_info() call failed", error);
    exit(1);
}
else
    quantum = sched_info.min_quantum;

/*
 * Fix the default processor set to take a fixed priority thread.
 */
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}
```

```
error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error != KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

error=processor_set_policy_enable(default_set_priv, POLICY_FIXEDPRI);
if (error != KERN_SUCCESS)
    mach_error("Error calling processor_set_policy_enable", error);

/*
 * Change the thread's scheduling policy to fixed priority.
 */
error=thread_policy(thread_self(), POLICY_FIXEDPRI, quantum);
if (error != KERN_SUCCESS)
    mach_error("thread_policy() call failed", error);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *thread* is not a thread, or *policy* is not a recognized policy.

KERN_FAILURE: The processor set to which *thread* is currently assigned doesn't permit *policy.*

SEE ALSO **processor_set_policy(), thread_switch()**

---

thread_priority(), thread_max_priority()

SUMMARY Set scheduling priority for thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_priority**(thread_t *thread*, int *priority*, boolean_t *set_max*)
kern_return_t **thread_max_priority**(thread_t *thread*, processor_set_t *processor_set*, int *priority*)

ARGUMENTS *thread*: The thread whose priority is to be changed.

priority: The new priority to change it to.

set_max: Also set *thread*'s maximum priority if true.

processor_set: The privileged port for the processor set to which *thread* is currently assigned.

DESCRIPTION Threads have three priorities associated with them by the system: a *base priority*, a *maximum priority*, and a *scheduled priority*.

The scheduled priority is used to make scheduling decisions about the thread. It's determined from the base priority by the policy. (For the timesharing and interactive policies, this means adding an increment derived from CPU usage). The base priority can be set under user control, but can never exceed the maximum priority. Raising the maximum priority requires presentation of the privileged port for the thread's processor set; since the privileged port for the default processor set is available only to the superuser, users cannot raise their maximum priority to unfairly compete with other users on that set. Newly created threads obtain their base priority from the task and their maximum priority from the thread.

Priorities range from 0 to 31, where higher numbers denote higher priorities. You can obtain the base, scheduled, and maximum priorities using **thread_info()**.

The **thread_priority()** function changes the base priority and optionally the maximum priority of *thread*. If the new base priority is higher than the scheduled priority of the currently executing thread, preemption may occur as a result of this call. The maximum priority of the thread is also set if *set_max* is true. This call fails if *priority* is greater than the current maximum priority of the thread. As a result, **thread_priority()** can lower—but never raise—the value of a thread's maximum priority.

The **thread_max_priority()** function changes the maximum priority of the thread. Because it requires the privileged port for the processor set, this call can reset the maximum priority to any legal value. If the new maximum priority is less than the thread's base priority, then the thread's base priority is set to the new maximum priority.

EXAMPLE

```
/* Get the privileged port for the default processor set. */
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error!=KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

/* Set the max priority. */
error=thread_max_priority(thread_self(), default_set_priv, priority);
if (error!=KERN_SUCCESS)
    mach_error("Call to thread_max_priority() failed",error);

/* Set the thread's priority. */
error=thread_priority(thread_self(), priority, FALSE);
if (error!=KERN_SUCCESS)
    mach_error("Call to thread_priority() failed",error);
```

RETURN KERN_SUCCESS: Operation completed successfully.

KERN_INVALID_ARGUMENT: *thread* is not a thread, *processor_set* is not a privileged port for a processor set, or *priority* is out of range (not in 0-31).

KERN_FAILURE: The requested operation would violate the thread's maximum (only for **thread_priority()**), or the thread is not assigned to the processor set whose privileged port was presented.

SEE ALSO **thread_policy(), task_priority(), processor_set_priority(), thread_switch()**

thread_reply() → See thread_get_special_port()

thread_resume()

SUMMARY Resume a thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_resume**(thread_t *target_thread*)

ARGUMENTS *target_thread*: The thread to be resumed.

DESCRIPTION The function **thread_resume()** decrements the thread's suspend count. If the count becomes 0, the thread is resumed. If it's still positive, the thread is left suspended. The suspend count never becomes negative.

RETURN KERN_SUCCESS: The thread has been resumed.

KERN_FAILURE: The suspend count is already 0.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread.

SEE ALSO **task_suspend()**, **task_resume()**, **thread_info()**, **thread_create()**, **thread_terminate()**, **thread_suspend()**

thread_self() → See thread_get_special_port()

thread_set_special_port() → See thread_get_special_port()

thread_set_state() → See thread_get_state()

thread_suspend()

SUMMARY Suspend a thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_suspend**(thread_t *target_thread*)

ARGUMENTS *target_thread*: The thread to be suspended.

DESCRIPTION The function **thread_suspend()** increments the thread's suspend count and prevents the thread from executing any more user-level instructions. In this context, a user-level instruction is either a machine instruction executed in user mode or a system trap instruction (including page faults).

If a thread is currently executing within a system trap, the kernel code may continue to execute until it reaches the system return code, or it may suspend within the kernel code. In either case, when the thread is resumed the system trap will return. This could cause unpredictable results if you did a suspend and then altered the user state of the thread in order to change its direction upon a resume. The function **thread_abort()** lets you abort any currently executing system call in a predictable way.

If the suspend count becomes greater than 1, it will take more than one **thread_resume()** call to restart the thread.

RETURN KERN_SUCCESS: The thread has been suspended.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread.

SEE ALSO **task_suspend()**, **task_resume()**, **thread_get_state()**, **thread_info()**, **thread_resume()**, **thread_terminate()**, **thread_abort()**

thread_switch()

SUMMARY Cause a context switch

SYNOPSIS

#import <mach/mach.h>

kern_return_t **thread_switch**(thread_t *new_thread*, int *option*, int *time*)

ARGUMENTS *new_thread*: Thread to switch to. If you specify THREAD_NULL, be sure to specify the *option* argument to be either SWITCH_OPTION_WAIT or SWITCH_OPTION_DEPRESS.

option: Specifies options associated with context switch. Three options are recognized:

SWITCH_OPTION_NONE: No options; the *time* argument is ignored. (You must set *new_thread* to a valid thread.)

SWITCH_OPTION_WAIT: This thread is blocked for the specified time. The block can be aborted by **thread_abort()**.

SWITCH_OPTION_DEPRESS: This thread's priority is depressed to the lowest possible value until one of the following happens: *time* milliseconds pass, this thread is scheduled again, or **thread_abort()** is called on this thread (whichever happens first). Priority depression is independent of operations that change this thread's priority; for example, **thread_priority()** will not abort the depression.

time: Time duration (in milliseconds) for options. The minimum time can be obtained as the **min_timeout** value from **host_info()**.

DESCRIPTION This function provides low-level access to the scheduler's context switching code. *new_thread* is a hint that implements handoff scheduling. The operating system will attempt to switch directly to *new_thread* (bypassing the normal logic that selects the next thread to run) if possible. If *new_thread* isn't valid or THREAD_NULL, **thread_switch()** returns an error.

The **thread_switch()** function is often called when the current thread can proceed no farther for some reason; the various options and arguments allow information about this reason to be transmitted to the kernel. The *new_thread* argument (handoff scheduling) is useful when the identity of the thread that must make progress before the current thread runs again is known. The SWITCH_OPTION_WAIT option is used when the amount of time that the current thread must wait before it can do anything useful can be estimated and is fairly long. The SWITCH_OPTION_DEPRESS option is used when the required waiting time is fairly short, especially when the identity of the thread that is being waited for is not known.

Users should beware of calling **thread_switch()** with an invalid *new_thread* (for example, THREAD_NULL) and no *option*. Because the timesharing and interactive schedulers vary the priority of threads based on usage, this may result in a waste of CPU time if the thread that must be run is of lower priority. The use of the SWITCH_OPTION_DEPRESS option in this situation is highly recommended.

When a thread that's depressed is scheduled, it regains its old priority. The code should recheck the conditions to see if it wants to depress again. If **thread_abort()** is called on a depressed thread, the priority of the thread is restored.

Users relying on the preemption semantics of a fixed-priority policy should be aware that **thread_switch()** ignores these semantics; it will run the specified *new_thread* independent of its priority and the priority of any other threads that could be run instead.

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *new_thread* is not a thread, or *option* is not a recognized option.

---

thread_terminate()

SUMMARY Terminate a thread

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **thread_terminate**(thread_t *target_thread*)

ARGUMENTS *target_thread*: The thread to be destroyed.

DESCRIPTION The function **thread_terminate()** destroys the thread specified by *target_thread.*

Warning: Don't use this function on threads that were created using the C-thread functions. Each C thread must terminate itself either explicitly, by calling **cthread_exit()**, or implicitly, by returning from its top-level function.

RETURN KERN_SUCCESS: The thread has been destroyed.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread.

SEE ALSO **task_terminate(), task_threads(), thread_create(), thread_resume(), thread_suspend()**

unix_pid()

SUMMARY Get the process ID of a task

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **unix_pid**(task_t *target_task*, int **pid*)

ARGUMENTS *target_task*: The task for which you want the process ID.

pid: Returns the process ID of *target_task*.

DESCRIPTION Returns the process ID of *target_task*. If the call doesn't succeed, *pid* is set to -1.

EXAMPLE

```
result=unix_pid(task_self(), &my_pid);
if (result!=KERN_SUCCESS) {
    mach_error("Call to unix_pid failed", result);
    exit(1);
}

printf("My process ID is %d\n", my_pid);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_FAILURE: *target_task* isn't a valid task. This might be because *target_task* is a pure Mach task (one created using **task_create**()).

SEE ALSO **task_by_unix_pid**()

vm_allocate()

SUMMARY Allocate virtual memory

SYNOPSIS

#import <mach/mach.h>

kern_return_t **vm_allocate**(vm_task_t *target_task*, vm_address_t **address*,
vm_size_t *size*, boolean_t *anywhere*)

ARGUMENTS

target_task: Task whose virtual memory is to be affected. Use **task_self()** to allocate memory in the caller's address space.

address: Starting address. If *anywhere* is true, the input value of this address will be ignored, and the space will be allocated wherever it's available. If *anywhere* is false, an attempt is made to allocate virtual memory starting at this virtual address. If this address isn't at the beginning of a virtual page, it gets rounded down so that it is. If there isn't enough space at this address, no memory will be allocated. No matter what the value of *anywhere* is, the address at which memory is actually allocated is returned in *address*.

size: Number of bytes to allocate (rounded up by the system to an integral number of virtual pages).

anywhere: If true, the kernel should find and allocate any region of the specified size. If false, virtual memory is allocated starting at *address* (rounded down to a virtual page boundary) if sufficient space exists.

DESCRIPTION

The function **vm_allocate()** allocates a region of virtual memory, placing it in the address space of the specified task. The physical memory isn't actually allocated until the new virtual memory is referenced. By default, the kernel rounds all addresses down to the nearest page boundary and all memory sizes up to the nearest page size. The global variable **vm_page_size** contains the page size. For languages other than C, the value of **vm_page_size** can be obtained by calling **vm_statistics()**.

Initially, the pages of allocated memory are protected to allow all forms of access, and are inherited in child tasks as a copy. Subsequent calls to **vm_protect()** and **vm_inherit()** may be used to change these properties. The allocated region is always zero-filled.

Note: Unless you have a special reason for calling **vm_allocate()** (such as a need for page-aligned memory), you should usually call **malloc()** or a similar C library function instead. The C library functions don't necessarily make UNIX or Mach system calls, so they're generally faster than using a Mach function such as **vm_allocate()**.

EXAMPLE

```
if ((ret = vm_allocate(task_self(), (vm_address_t *)&lock,
    sizeof(int), TRUE)) != KERN_SUCCESS) {
    mach_error("vm_allocate returned value of ", ret);
    printf("Exiting with error.\n");
    exit(-1);
}
if ((ret = vm_inherit(task_self(), (vm_address_t)lock, sizeof(int),
        VM_INHERIT_SHARE)) != KERN_SUCCESS) {
    mach_error("vm_inherit returned value of ", ret);
    printf("Exiting with error.\n");
    exit(-1);
}
```

RETURN

KERN_SUCCESS: Memory allocated.

KERN_INVALID_ADDRESS: Illegal address specified.

KERN_NO_SPACE: Not enough space left to satisfy this request.

SEE ALSO **vm_deallocate(), vm_inherit(), vm_protect(), vm_region(), vm_statistics()**

---

vm_copy()

SUMMARY Copy virtual memory

SYNOPSIS

#import <mach/mach.h>

kern_return_t **vm_copy**(vm_task_t *target_task*, vm_address_t *source_address*, vm_size_t *size*, vm_address_t *dest_address*)

ARGUMENTS

target_task: The task whose virtual memory is to be affected.

source_address: The address in *target_task* of the start of the source range (must be a page boundary).

size: The number of bytes to copy (must be a multiple of **vm_page_size**).

dest_address: The address in *target_task* of the start of the destination range (must be a page boundary).

DESCRIPTION The function **vm_copy()** causes the source memory range to be copied to the destination address; the destination region must not overlap the source region. The destination address range must already be allocated and writable; the source range must be readable.

For languages other than C, the value of **vm_page_size** can be obtained by calling **vm_statistics()**.

EXAMPLE

```
if ((rtn = vm_allocate(task_self(), (vm_address_t *)&data1,
        vm_page_size, TRUE)) != KERN_SUCCESS) {
    mach_error("vm_allocate returned value of ", rtn);
    printf("vm_copy: Exiting.\n");
    exit(-1);
}

temp = data1;
for (i = 0; (i < vm_page_size / sizeof(int)); i++)
    temp[i] = i;
printf("vm_copy: set data\n");

if ((rtn = vm_allocate(task_self(), (vm_address_t *)&data2,
        vm_page_size, TRUE)) != KERN_SUCCESS) {
    mach_error("vm_allocate returned value of ", rtn);
    printf("vm_copy: Exiting.\n");
    exit(-1);
}

if ((rtn = vm_copy(task_self(), (vm_address_t)data1, vm_page_size,
        (vm_address_t)data2)) != KERN_SUCCESS) {
    mach_error("vm_copy returned value of ", rtn);
    printf("vm_copy: Exiting.\n");
    exit(-1);
}
```

RETURN KERN_SUCCESS: Memory copied.

KERN_INVALID_ARGUMENT: The address doesn't start on a page boundary or the size isn't a multiple of **vm_page_size**.

KERN_PROTECTION_FAILURE: The destination region isn't writable, or the source region isn't readable.

KERN_INVALID_ADDRESS: An illegal or nonallocated address was specified, or insufficient memory was allocated at one of the addresses.

SEE ALSO **vm_allocate(), vm_protect(), vm_write(), vm_statistics()**

vm_deallocate()

SUMMARY Deallocate virtual memory

SYNOPSIS

#import <mach/mach.h>

kern_return_t **vm_deallocate**(vm_task_t *target_task*, vm_address_t *address*, vm_size_t *size*)

ARGUMENTS

target_task: Task whose virtual memory is to be affected.

address: Starting address (this gets rounded down to a page boundary).

size: Number of bytes to deallocate (this gets rounded up to a page boundary).

DESCRIPTION The function **vm_deallocate()** relinquishes access to a region of a task's address space, causing further access to that memory to fail. This address range will be available for reallocation. Note that because of the rounding to virtual page boundaries, more than *size* bytes may be deallocated. Use **vm_statistics()** or the global variable **vm_page_size** to get the current virtual page size.

This function may be used to deallocate memory that was passed to a task in a message (using out-of-line data). In that case, the rounding should cause no trouble, since the region of memory was allocated as a set of pages.

The function **vm_deallocate()** affects only the task specified by *target_task*. Other tasks that may have access to this memory can continue to reference it.

EXAMPLE

```
r = vm_deallocate(task_self(), (vm_address_t)thread_list,
    sizeof(thread_list)*thread_count);
if (r != KERN_SUCCESS)
    mach_error("Trouble freeing thread_list", r);
```

RETURN KERN_SUCCESS: Memory deallocated.

KERN_INVALID_ADDRESS: Illegal or nonallocated address specified.

SEE ALSO **vm_allocate()**, **vm_statistics()**, **msg_receive()**

vm_inherit()

SUMMARY Inherit virtual memory

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **vm_inherit**(vm_task_t *target_task*, vm_address_t *address*, vm_size_t *size*, vm_inherit_t *new_inheritance*)

ARGUMENTS *target_task*: Task whose virtual memory is to be affected.

address: Starting address (this gets rounded down to a page boundary).

size: Size in bytes of the region for which inheritance is to change (this gets rounded up to a page boundary).

new_inheritance: How this memory is to be inherited in child tasks. Inheritance is specified by using one of these following three values:

VM_INHERIT_SHARE: Child tasks will share this memory with this task.
VM_INHERIT_COPY: Child tasks will receive a copy of this region.
VM_INHERIT_NONE: This region will be absent from child tasks.

DESCRIPTION The function **vm_inherit()** specifies how a region of a task's address space is to be passed to child tasks at the time of task creation. Inheritance is an attribute of virtual pages; thus the addresses and size of memory to be set will be rounded to refer to whole pages.

Setting **vm_inherit()** to VM_INHERIT_SHARE and forking a child task is the only way two Mach tasks can share physical memory. However, all the threads of a given task share all the same memory.

EXAMPLE

```
if ((ret = vm_allocate(task_self(), (vm_address_t *)&lock, sizeof(int),
        TRUE)) != KERN_SUCCESS) {
    mach_error("vm_allocate returned value of ", ret);
    printf("Exiting with error.\n");
    exit(-1);
}
if ((ret = vm_inherit(task_self(), (vm_address_t)lock, sizeof(int),
        VM_INHERIT_SHARE)) != KERN_SUCCESS) {
    mach_error("vm_inherit returned value of ", ret);
    printf("Exiting with error.\n");
    exit(-1);
}
```

RETURN KERN_SUCCESS: The inheritance has been set.

KERN_INVALID_ADDRESS: Illegal address specified.

SEE ALSO **task_create()**, **vm_region()**

---

vm_protect()

SUMMARY Protect virtual memory

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **vm_protect**(vm_task_t *target_task*, vm_address_t *address*, vm_size_t *size*, boolean_t *set_maximum*, vm_prot_t *new_protection*)

ARGUMENTS *target_task*: Task whose virtual memory is to be affected.

address: Starting address (this gets rounded down to a page boundary).

size: Size in bytes of the region for which protection is to change (this gets rounded up to a page boundary).

set_maximum: If set, make the protection change apply to the maximum protection associated with this address range; otherwise, change the current protection on this range. If the maximum protection is reduced below the current protection, both will be changed to reflect the new maximum.

new_protection: A new protection value for this region; either VM_PROT_NONE or some combination of VM_PROT_READ, VM_PROT_WRITE, and VM_PROT_EXECUTE.

DESCRIPTION The function **vm_protect()** changes the protection of some pages of allocated memory in a task's address space. In general, a protection value permits the named operation. When memory is first allocated it has all protection bits on. The exact interpretation of a protection value is machine-dependent. In the NeXT Mach operating system, three levels of memory protection are provided:

- No access
- Read and execute access
- Read, execute, and write access

VM_PROT_NONE permits no access. VM_PROT_WRITE permits read, execute, and write access; VM_PROT_READ or VM_PROT_EXECUTE permits read and execute access, but not write access.

EXAMPLE

```
vm_address_t    addr = (vm_address_t)mlock;

r = vm_protect(task_self(), addr, vm_page_size, FALSE, 0);
if (r != KERN_SUCCESS) {
    mach_error("vm_protect 0", r);
    exit(1);
}
printf("protect on\n");
```

RETURN KERN_SUCCESS: The memory has been protected.

KERN_PROTECTION_FAILURE: An attempt was made to increase the current or maximum protection beyond the existing maximum protection value.

KERN_INVALID_ADDRESS: An illegal or nonallocated address was specified.

---

vm_read()

SUMMARY Read virtual memory

SYNOPSIS

#import <mach/mach.h>

kern_return_t **vm_read**(vm_task_t *target_task*, vm_address_t *address*, vm_size_t *size*, pointer_t **data*, unsigned int **data_count*)

ARGUMENTS

target_task: Task whose memory is to be read.

address: The first address to be read (must be on a page boundary).

size: The number of bytes of data to be read (must be a multiple of **vm_page_size**).

data: The array of data copied from the given task.

data_count: Returns the size of the data array in bytes (will be an integral number of pages).

DESCRIPTION The function **vm_read()** allows one task's virtual memory to be read by another task. The data array is returned in a newly allocated region; the task reading the data should call **vm_deallocate()** on this region when it's done with the data.

For languages other than C, the value of **vm_page_size** can be obtained by calling **vm_statistics()**.

EXAMPLE

```
if ((rtn = vm_allocate(task_self(), (vm_address_t *)&data1,
        vm_page_size, TRUE)) != KERN_SUCCESS) {
    mach_error("vm_allocate returned value of ", rtn);
    printf("vmread: Exiting.\n");
    exit(-1);
}

temp = data1;
for (i = 0; (i < vm_page_size); i++)
    temp[i] = i;
printf("Filled space allocated with some data.\n");
printf("Doing vm_read....\n");
if ((rtn = vm_read(task_self(), (vm_address_t)data1, vm_page_size,
        (pointer_t *)&data2, &data_cnt)) != KERN_SUCCESS) {
    mach_error("vm_read returned value of ", rtn);
    printf("vmread: Exiting.\n");
    exit(-1);
}
printf("Successful vm_read.\n");
```

RETURN KERN_SUCCESS: The memory has been read.

KERN_INVALID_ARGUMENT: Either *address* does not start on a page boundary or *size* isn't an integral number of pages.

KERN_NO_SPACE: There isn't enough room in the caller's virtual memory to allocate space for the data to be returned.

KERN_PROTECTION_FAILURE: The address region in the target task is protected against reading.

KERN_INVALID_ADDRESS: An illegal or nonallocated address was specified, or there were not *size* bytes of data following that address.

SEE ALSO **vm_write(), vm_copy(), vm_deallocate()**

vm_region()

SUMMARY Get information about virtual memory regions

SYNOPSIS

#import <mach/mach.h>

kern_return_t **vm_region**(vm_task_t *target_task*, vm_address_t **address*, vm_size_t **size*, vm_prot_t **protection*, vm_prot_t **max_protection*, vm_inherit_t **inheritance*, boolean_t **shared*, port_t **object_name*, vm_offset_t **offset*)

ARGUMENTS

target_task: The task for which an address space description is requested.

address: The address at which to start looking for a region. On return, *address* will contain the start of the region (therefore, the value returned will be different from the value that was passed in if the specified region is part of a larger region).

size: Returns the size (in bytes) of the located region.

protection: Returns the current protection of the region.

max_protection: Returns the maximum allowable protection for this region.

inheritance: Returns the inheritance attribute for this region.

shared: Returns true if this region is shared, false if it isn't.

object_name: Returns the port identifying the region's memory object.

offset: Returns the offset into the pager object at which this region begins.

DESCRIPTION

The **vm_region()** function returns a description of the specified region of the target task's virtual address space. This function begins at *address*, looking forward through memory until it comes to an allocated region. (If *address* is in a region, that region is used.) If *address* isn't in a region, it's set to the start of the first region that follows the incoming value. In this way an entire address space can be scanned. You can set *address* to the constant VM_MIN_ADDRESS (defined in the header file **mach/machine/vm_param.h**) to specify the first address in the address space.

EXAMPLE

```
char            *data;
kern_return_t   r;
vm_size_t       size;
vm_prot_t       protection, max_protection;
vm_inherit_t    inheritance;
boolean_t       shared;
port_t          object_name;
vm_offset_t     offset;
vm_address_t    address;

/* . . . */
/* Check the inheritance of "data". */
address = (vm_address_t)&data;
r = vm_region(task_self(), &address, &size, &protection,
    &max_protection, &inheritance, &shared, &object_name, &offset);

if (r != KERN_SUCCESS)
    mach_error("Error calling vm_region", r);
else {
    printf("Inheritance is:  ");
    switch (inheritance) {
        case VM_INHERIT_SHARE:
            printf("Share with child\n");
            break;
        case VM_INHERIT_COPY:
            printf("Copy into child\n");
            break;
        case VM_INHERIT_NONE:
            printf("Absent from child\n");
            break;
        case VM_INHERIT_DONATE_COPY:
            printf("Copy and delete\n");
            break;
    }
}
```

RETURN KERN_SUCCESS: The region was located and information has been returned.

KERN_NO_SPACE: The task contains no region at or above *address*.

SEE ALSO **vm_allocate(), vm_deallocate(), vm_protect(), vm_inherit()**

vm_statistics()

SUMMARY Examine virtual memory statistics

SYNOPSIS **#import <mach/mach.h>**

kern_return_t **vm_statistics**(vm_task_t *target_task*, vm_statistics_data_t **vm_stats*)

ARGUMENTS *target_task*: The task that's requesting the statistics.

vm_stats: Returns the statistics.

DESCRIPTION The function **vm_statistics()** returns statistics about the kernel's use of virtual memory since the kernel was booted. The system page size is contained in both the **pagesize** field of the *vm_status* and the global variable **vm_page_size**, which is set at task initialization and remains constant for the life of the task.

```
struct vm_statistics {
    long  pagesize;         /* page size in bytes */
    long  free_count;       /* number of pages free */
    long  active_count;     /* number of pages active */
    long  inactive_count;   /* number of pages inactive */
    long  wire_count;       /* number of pages wired down */
    long  zero_fill_count;  /* number of zero-fill pages */
    long  reactivations;    /* number of pages reactivated */
    long  pageins;          /* number of pageins */
    long  pageouts;         /* number of pageouts */
    long  faults;           /* number of faults */
    long  cow_faults;       /* number of copy-on-writes */
    long  lookups;          /* object cache lookups */
    long  hits;             /* object cache hits */
};
typedef struct vm_statistics   vm_statistics_data_t;
```

EXAMPLE

```
result=vm_statistics(task_self(), &vm_stats);
if (result != KERN_SUCCESS)
    mach_error("An error calling vm_statistics()!", result);
else
    printf("%d bytes of RAM are free\n",
        vm_stats.free_count * vm_stats.pagesize);
```

RETURN KERN_SUCCESS: The operation was successful.

vm_write()

SUMMARY Write virtual memory

SYNOPSIS

#import <mach/mach.h>

kern_return_t **vm_write**(vm_task_t *target_task*, vm_address_t *address*, pointer_t *data*, unsigned int *data_count*)

ARGUMENTS

target_task: Task whose memory is to be written.

address: Starting address in task to be affected (must be a page boundary).

data: An array of bytes to be written.

data_count: The size in bytes of the data array (must be a multiple of **vm_page_size**).

DESCRIPTION The function **vm_write()** allows a task's virtual memory to be written by another task. For languages other than C, the value of **vm_page_size** can be obtained by calling **vm_statistics()**.

RETURN KERN_SUCCESS: Memory written.

KERN_INVALID_ARGUMENT: The address doesn't start on a page boundary, or the size isn't an integral number of pages.

KERN_PROTECTION_FAILURE: The address region in the target task is protected against writing.

KERN_INVALID_ADDRESS: An illegal or nonallocated address was specified or the amount of allocated memory starting at *address* was less than *data_count*.

SEE ALSO **vm_copy()**, **vm_protect()**, **vm_read()**, **vm_statistics()**

Bootstrap Server Functions

The Bootstrap Server, like the Network Name Server, lets tasks publish ports that other tasks can send messages to. Unlike the Network Name Server, the Bootstrap Server is designed so that each server and its clients must be on the same host. The Bootstrap Server accomplishes this by using each task's bootstrap port (which is inherited from its parent) to ensure that the task is a descendent of a local task.

When a task forks a child task that shouldn't have access to the same set of services as the parent, the parent task must change its own bootstrap port—perhaps only temporarily—so that its child inherits a *subset port*. The parent should then change the set of services available on the subset port to suit the child's requirements.

The Bootstrap Server was created by NeXT, so these functions aren't in other versions of Mach. See **/NextDeveloper/Headers/servers/bootstrap.defs** for more information of how the Bootstrap Server works.

Note: If possible, you should use Distributed Objects instead of the Bootstrap Server. The Distributed Objects system is described in the *NeXTSTEP General Reference*.

---

bootstrap_check_in()

SUMMARY Get receive rights to a service port

SYNOPSIS

#import <mach/mach.h>
#import <servers/bootstrap.h>

kern_return_t **bootstrap_check_in**(port_t *bootstrap_port*, name_t *service_name*,
port_all_t **service_port*)

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

service_name: The string that names the service.

service_port: Returns receive rights to the service port.

DESCRIPTION Use this function in a server to start providing a service. The service must already be defined, either by the appropriate line in **/etc/bootstrap.conf** or by a call to **bootstrap_create_service()**. Calling **bootstrap_check_in()** makes the service active.

EXAMPLE

```
/* Get receive rights for our service. */
result=bootstrap_check_in(bootstrap_port, MYNAME, &my_service_port);
if (result != BOOTSTRAP_SUCCESS)
    mach_error("Couldn't create service", result);
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NOT_PRIVILEGED: *bootstrap_port* is an unprivileged bootstrap port.

BOOTSTRAP_UNKNOWN_SERVICE: The service doesn't exist. It might be defined in a subset (see **bootstrap_subset()**).

BOOTSTRAP_SERVICE_ACTIVE: The service has already been registered or checked in and the server hasn't died.

Returns appropriate kernel errors on RPC failure.

---

bootstrap_create_service()

SUMMARY Create a service and service port

SYNOPSIS

#import <mach/mach.h>
#import <servers/bootstrap.h>

kern_return_t **bootstrap_create_service**(port_t *bootstrap_port*, name_t *service_name*, port_t **service_port*)

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

service_name: The string that specifies the service.

service_port: Returns send rights for the service.

DESCRIPTION Creates a service named *service_name* and returns send rights to that port in *service_port*. The port may later be checked in as if this port were configured in the bootstrap configuration file. (At that time **bootstrap_check_in()** will return receive rights to *service_port* and will make the service active.)

This function is often used to create services that are available only to a subset of tasks (see **bootstrap_subset()**). Any task can call this function—it doesn't have to be the server.

EXAMPLE

```
/* Tell the bootstrap server about a service. */
result=bootstrap_create_service(bootstrap_port, SERVICENAME,
    &service_port);
if (result!=BOOTSTRAP_SUCCESS)
    mach_error("Couldn't create service", result);
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NOT_PRIVILEGED: *bootstrap_port* is an unprivileged bootstrap port.

BOOTSTRAP_SERVICE_ACTIVE: The service already exists.

Returns appropriate kernel errors on RPC failure.

---

bootstrap_info()

SUMMARY Get information about all known services

SYNOPSIS

#import <mach/mach.h>
#import <servers/bootstrap.h>

kern_return_t **bootstrap_info**(port_t *bootstrap_port*, name_array_t **service_names*,
unsigned int **service_names_count*, name_array_t **server_names*,
unsigned int **server_names_count*, bool_array_t **service_active*,
unsigned int **service_active_count*)

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

service_names: Returns the names of all known services.

service_names_count: Returns the number of service names.

server_names: Returns the name, if known, of the server that provides the corresponding service. Except for the **mach_init** server, this name isn't known unless the bootstrap configuration file has a **server** line for this server.

server_names_count: Returns the number of server names.

service_active: Returns an array of booleans that correspond to the *service_names* array. For each item, the boolean value is true if the service is receiving messages sent to its port; otherwise, it's false.

service_active_count: Returns the number of items in the *service_active* array.

DESCRIPTION This function returns information about all services that are known. Note that it won't return information on services that are defined only in subsets, unless the subset port is an ancestor of *bootstrap_port*. (See **bootstrap_subset()** for information on subsets.)

EXAMPLE

```
result = bootstrap_info(bootstrap_port, &service_names, &service_cnt,
  &server_names, &server_cnt, &service_active, &service_active_cnt);
if (result != BOOTSTRAP_SUCCESS)
    printf("ERROR:  info failed: %d", result);
else {
    for (i = 0; i < service_cnt; i++)
        printf("Name: %-15s   Server: %-15s    Active: %-4s",
            service_names[i],
            server_names[i][0] == '\0' ? "Unknown" : server_names[i],
            service_active[i] ? "Yes\n" : "No\n");
}
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NO_MEMORY: The Bootstrap Server couldn't allocate enough memory to return the information.

Returns appropriate kernel errors on RPC failure.

bootstrap_look_up()

SUMMARY Get the service port of a particular service

SYNOPSIS

#import <mach/mach.h>
#import <servers/bootstrap.h>

kern_return_t **bootstrap_look_up**(port_t *bootstrap_port*, name_t *service_name*,
port_t **service_port*)

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

service_name: The string that identifies the service.

service_port: Returns send rights for the service port.

DESCRIPTION Returns send rights for the service port of the specified service. The service isn't guaranteed to be active. (To check whether the service is active, use **bootstrap_status()**.)

EXAMPLE

```
result=bootstrap_look_up(bootstrap_port, "FreeService2", &srvc_port);
if (result!=BOOTSTRAP_SUCCESS)
    printf("lookup failed: %d\n", result);
else {
    /* Access the service by sending messages to srvc_port. */
}
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_UNKNOWN_SERVICE: The service doesn't exist. It might be defined in a subset (see **bootstrap_subset()**).

Returns appropriate kernel errors on RPC failure.

bootstrap_look_up_array()

SUMMARY Get the service ports for an array of services

SYNOPSIS

#import <mach/mach.h>
#import <servers/bootstrap.h>

kern_return_t **bootstrap_look_up_array**(port_t *bootstrap_port*,
name_array_t *service_names*, unsigned int *service_names_count*,
port_array_t **service_ports*, unsigned int **service_ports_count*,
boolean_t **all_services_known*)

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

service_names: An array of service names.

service_names_count: The number of service names.

service_ports: Returns an array of service ports.

service_ports_count: Returns the number of service ports. This should be equal to *service_names_count*.

all_services_known: Returns true if every service name was recognized; otherwise returns false.

DESCRIPTION Returns port send rights in corresponding entries of the array *service_ports* for all services named in the array *service_names*. You should call **vm_deallocate()** on *service_ports* when you no longer need it.

Unknown service names have the corresponding service port set to PORT_NULL. Note that these services might be available in a subset (see **bootstrap_subset()**).

EXAMPLE

```
kern_return_t   result;
port_t          my_bootstrap_port;
unsigned int    port_cnt;
boolean_t       all_known;
name_t          name_array[2]={"Service", "NetMessage"};
port_array_t    ports;
```

```
result = task_get_bootstrap_port(task_self(), &my_bootstrap_port);
if (result != KERN_SUCCESS) {
    mach_error("Couldn't get bootstrap port", result);
    exit(1);
}

result=bootstrap_look_up_array(my_bootstrap_port, name_array, 2,
    &ports, &port_cnt, &all_known);
if (result!=BOOTSTRAP_SUCCESS)
    mach_error("Lookup array failed", result);
else
    printf("Port count = %d, all known = %d\n", port_cnt, all_known);

/* . . . */
result=vm_deallocate(task_self(), (vm_address_t)ports,
    sizeof(ports)*port_cnt);
if (result != KERN_SUCCESS)
    mach_error("Trouble freeing ports", result);
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_BAD_COUNT: *service_names_count* was too large (greater than BOOTSTRAP_MAX_LOOKUP_COUNT, which is defined in the header file **server/bootstrap_defs.h**).

Returns appropriate kernel errors on RPC failure.

---

bootstrap_register()

SUMMARY Register send rights for a service port

SYNOPSIS

#import <mach/mach.h>
#import <servers/bootstrap.h>

kern_return_t **bootstrap_register**(port_t *bootstrap_port*, name_t *service_name*, port_t *service_port*)

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

service_name: The string that identifies the service.

service_port: The service port for the service.

DESCRIPTION You can use this function to create a server that hasn't been defined in the bootstrap configuration file. This function specifies to the Bootstrap Server exactly which port should be the service port.

You can't register a service if an active binding already exists. However, you can register a service if the existing binding is inactive (that is, the Bootstrap Server currently holds receive rights for the service port); in this case the previous service port will be deallocated.

A service that is restarting can resume service for previous clients by setting *service_port* to the previous service port. You can get this port by calling **bootstrap_check_in()**.

EXAMPLE

```
/* Create a port to use as the service port. */
result=port_allocate(task_self(), &myport);
if (result != KERN_SUCCESS) {
    mach_error("Couldn't allocate a service port", result);
    exit(1);
}

/* Tell the bootstrap server about my service. */
result=bootstrap_register(bootstrap_port, MYNAME, myport);
if (result != BOOTSTRAP_SUCCESS)
    printf("Call to bootstrap_register failed: %d", result);
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NOT_PRIVILEGED: *bootstrap_port* is an unprivileged bootstrap port.

BOOTSTRAP_NAME_IN_USE: The service is already active.

Returns appropriate kernel errors on RPC failure.

bootstrap_status()

SUMMARY Check whether a service is available

SYNOPSIS

#import <mach/mach.h>
#import <servers/bootstrap.h>

kern_return_t **bootstrap_status**(port_t *bootstrap_port*, name_t *service_name*,
boolean_t **service_active*)

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

service_name: The string that specifies a particular service.

service_active: Returns true if the service is active; otherwise, returns false.

DESCRIPTION This function tells you whether a service is known to users of *bootstrap_port*, and whether it's active. A service is active if a server is able to receive messages on its service port. If a service isn't active, the Bootstrap Server holds receive rights for the service port.

EXAMPLE

```
result=bootstrap_status(bootstrap_port, MYNAME, &service_active);
if (result!=BOOTSTRAP_SUCCESS)
    printf("status check failed\n");
else {
    if (service_active)
        printf("Server %s is active\n", MYNAME);
    else
        printf ("Server %s is NOT active\n", MYNAME);
}
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_UNKNOWN_SERVICE: The service doesn't exist. It might be defined in a subset (see **bootstrap_subset()**).

Returns appropriate kernel errors on RPC failure.

bootstrap_subset()

SUMMARY Get a new port to use as a bootstrap port

SYNOPSIS

#import <mach/mach.h>
#import <servers/bootstrap.h>

kern_return_t **bootstrap_subset**(port_t *bootstrap_port*, port_t *requestor_port*,
port_t **subset_port*)

ARGUMENTS *bootstrap_port*: A bootstrap port. Usually, this should be the task's default bootstrap port, which is returned by **task_get_bootstrap_port()**.

requestor_port: A port that determines the life span of the subset.

subset_port: Returns the subset port.

DESCRIPTION Returns a new port to use as a bootstrap port. This port behaves exactly like the previous *bootstrap_port*, with one exception: When you register a port by calling **bootstrap_register()** using *subset_port* as the bootstrap port, the registered port is available only to users of *subset_port* and its descendants. Lookups on the *subset_port* will return ports registered specifically with this port, and will also return ports registered with ancestors of this *subset_port*. (The ancestors of *subset_port* are *bootstrap_port* and, if *bootstrap_port* is itself a subset port, any ancestors of *bootstrap_port*.)

You can override a service already registered with an ancestor port by registering it with the subset port. Any thread that looks up the service using the subset port will see only the version of the service that's registered with the subset port. This is one way to transparently provide services such as monitor programs or individualized spelling checkers, while the rest of the system still uses the default service.

When it's detected that *requestor_port* is destroyed, the subset port and its descendants are destroyed; the services advertised by these ports are destroyed, as well.

```
/* Get and save the current bootstrap port for this task. */
r = task_get_bootstrap_port(task_self(), &old_bs_port);
if (r != KERN_SUCCESS) {
    mach_error("task_get_bootstrap_port", r);
    exit(1);
}
```

```
/* Get a subset port. */
r = bootstrap_subset(old_bs_port, task_self(), &subset_port);
if (r != BOOTSTRAP_SUCCESS) {
    mach_error("Couldn't get unpriv port", r);
    exit(1);
}

/* Set the bootstrap port */
r = task_set_bootstrap_port(task_self(), subset_port);
if (r != KERN_SUCCESS) {
    mach_error("task_set_bootstrap_port", r);
    exit(1);
}
bootstrap_port = subset_port;
```

RETURN BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NOT_PRIVILEGED: *bootstrap_port* is an unprivileged bootstrap port.

Returns appropriate kernel errors on RPC failure.

Network Name Server Functions

If possible, you should use Distributed Objects instead of the Network Name Server functions. The Distributed Objects system is described in the *NeXTSTEP General Reference*.

---

netname_check_in()

SUMMARY Check a name into the local name space

SYNOPSIS

#import <mach/mach.h>
#import <servers/netname.h>

kern_return_t **netname_check_in**(port_t *server_port*, netname_name_t *port_name*, port_t *signature*, port_t *port_id*)

ARGUMENTS *server_port*: The task's port to the Network Name Server. To use the system Network Name Server, this should be set to the global variable **name_server_port**.

port_name: The name of the port to be checked in.

signature: The port used to protect the right to remove a name.

port_id: The port to be checked in.

DESCRIPTION The function **netname_check_in()** enters a port with the name *port_name* into the name space of the local network server. The *signature* argument is a port that's used to protect this name. This same port must be presented on a **netname_check_out()** call for that call to be able to remove the name from the name space.

RETURN NETNAME_SUCCESS: The operation succeeded.

SEE ALSO **netname_check_out(), netname_look_up()**

netname_check_out()

SUMMARY Remove a name from the local name space

SYNOPSIS

#import <mach/mach.h>
#import <servers/netname.h>

kern_return_t **netname_check_out**(port_t *server_port*, netname_name_t *port_name*, port_t *signature*)

ARGUMENTS *server_port*: The task's port to the Network Name Server. To use the system Network Name Server, this should be set to **name_server_port**.

port_name: The name of the port to be checked out.

signature: The port used to protect the right to remove a name.

DESCRIPTION The function **netname_check_out()** removes a port with the name *port_name* from the name space of the local network server. The *signature* argument must be the same port as the signature port passed to **netname_check_in()** when this name was checked in.

RETURN NETNAME_SUCCESS: The operation succeeded.

NETNAME_NOT_YOURS: The signature given to **netname_check_out()** did not match the signature with which the port was checked in.

SEE ALSO **netname_check_in()**, **netname_look_up()**

---

netname_look_up()

SUMMARY Look up a name on a specific host

SYNOPSIS

#import <mach/mach.h>
#import <servers/netname.h>

kern_return_t **netname_look_up**(port_t *server_port*, netname_name_t *host_name*, netname_name_t *port_name*, port_t **port_id*)

ARGUMENTS *server_port*: The task's port to the Network Name Server. To use the system Network Name Server, this should be set to **name_server_port**.

host_name: The name of the host to query. This can't be a null pointer.

port_name: The name of port to be looked up.

port_id: The port that was looked up.

DESCRIPTION The function **netname_look_up()** returns the value of the port named by *port_name* by questioning the host named by the *host_name* argument. Thus this call is a directed name lookup. The *host_name* may be any of the host's official nicknames. If it's an empty string, the local host is assumed. If *host_name* is "*", a broadcast lookup is performed.

Important: Use **NXPortNameLookup()** instead of **netname_look_up()** in all NeXTSTEP applications. (In the future, Listener instances might register with a server other than the Network Name Server.)

RETURN NETNAME_SUCCESS: The operation succeeded.

NETNAME_NOT_CHECKED_IN: **netname_look_up()** could not find the name at the given host.

NETNAME_NO_SUCH_HOST: The *host_name* argument to **netname_look_up()** does not name a valid host.

NETNAME_HOST_NOT_FOUND: **netname_look_up()** could not reach the host named by *host_name* (for instance, because it's down).

SEE ALSO **netname_check_in(), netname_check_out()**

Kernel-Server Loader Functions

To use these functions, you must compile with the kernload library. For example:

```
cc myprog.c -lkernload
```

---

kern_loader_abort()

SUMMARY Shut down or reconfigure **kern_loader**

SYNOPSIS

#import <mach/mach.h>
#import <kernserv/kern_loader.h>

kern_return_t **kern_loader_abort**(port_t *loader_port*, port_t *priv_port*, boolean_t *restart*)

ARGUMENTS

loader_port: **kern_loader** port, obtained from **kern_loader_look_up()**.

priv_port: The privileged port for this host, returned by **host_priv_self()**.

restart: If true, reconfigure **kern_loader**.

DESCRIPTION This function unloads and deallocates all loadable kernel servers and then, depending on the value of *restart*, kills or reconfigures the kernel-server loader. If *restart* is true, then **kern_loader** rereads its configuration file (**etc/kern_loader.conf**) to determine which servers it should allocate and load.

EXAMPLE

```
/* Get kern_loader's port. */
error=kern_loader_look_up(&loader_port);
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't find kern_loader's port", error);
    exit(1);
}

/* Reconfigure kern_loader. */
error=kern_loader_abort(loader_port, host_priv_self(), TRUE);
if (error != KERN_SUCCESS)
    kern_loader_error("Couldn't stop kern_loader", error);
```

RETURN KERN_SUCCESS: The call was successful.

KERN_LOADER_NO_PERMISSION: *priv_port* isn't the host's privileged port. (Make sure **host_priv_self()** is called by a process with superuser permission.)

SEE ALSO **kern_loader_delete_server()**, **kern_loader_look_up()**, **kern_loader_unload_server()**

---

kern_loader_add_server()

SUMMARY Allocate a loadable kernel server

SYNOPSIS

```
#import <mach/mach.h>
#import <kernserv/kern_loader.h>
```

kern_return_t **kern_loader_add_server**(port_t *loader_port*, port_t *task_port*, server_reloc_t *server_reloc*)

ARGUMENTS *loader_port*: **kern_loader** port, obtained from calling **kern_loader_look_up()**.

task_port: The kernel's task port, obtained using **task_by_unix_pid()**.

server_reloc: The server's relocatable object file. For example, the relocatable object file of the MIDI driver is "/usr/lib/kern_loader/Midi/mididriver_reloc".

DESCRIPTION This function prepares the loadable kernel server to be loaded into the kernel. The server isn't loaded unless it automatically loads when allocated.

If the server is already loaded or allocated, then the server is unloaded (if necessary) and allocated again from scratch.

EXAMPLE

```
/* Get kern_loader's port. */
r = kern_loader_look_up(&loader_port);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't get loader_port", r);
    exit(1);
}
```

```
/* Get the kernel's task port. */
r = task_by_unix_pid(task_self(), 0, &kernel_task);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't get kernel_task", r);
    exit(2);
}

/* Add the server. */
r = kern_loader_add_server(loader_port, kernel_task,
    "/usr/lib/kern_loader/Midi/midi_reloc");
if (r != KERN_SUCCESS) {
    kern_loader_error("Call to kern_loader_abort failed", r);
    exit(3);
}
```

RETURN KERN_SUCCESS: The server has been successfully allocated.

KERN_LOADER_NO_PERMISSION: *task_port* wasn't the kernel's task port. (Make sure **task_by_unix_pid()** is called by a process with superuser permission.)

KERN_LOADER_SERVER_WONT_LOAD: The kernel-server loader couldn't use *server_reloc* to build an loadable object file, or it couldn't understand the load or unload commands, or it couldn't link the loadable object file against **/mach**.

SEE ALSO **kern_loader_delete_server()**, **kern_loader_look_up()**

---

kern_loader_delete_server()

SUMMARY Delete a loadable kernel server

SYNOPSIS

#import <mach/mach.h>
#import <kernserv/kern_loader.h>

kern_return_t **kern_loader_delete_server**(port_t *loader_port*, port_t *task_port*, server_name_t *server_name*)

ARGUMENTS *loader_port*: **kern_loader** port, obtained from calling **kern_loader_look_up()**.

task_port: The kernel's task port, obtained using **task_by_unix_pid()**.

server_name: The string associated with the server. For example, the name of the MIDI driver is "mididriver".

DESCRIPTION This function removes the loadable kernel server from **kern_loader** control. If the server is currently loaded, then it's unloaded.

EXAMPLE

```
/* Get kern_loader's port. */
error=kern_loader_look_up(&loader_port);
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't find kern_loader's port", error);
    exit(1);
}

/* Get the kernel's task port. */
error=task_by_unix_pid(task_self(), 0, &kern_port);
if (error != KERN_SUCCESS) {
    mach_error("Error looking up kernel port", error);
    exit(2);
}

/* Delete the server. */
error=kern_loader_delete_server(loader_port, kern_port, "mididriver");
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't delete mididriver", error);
    exit(3);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_LOADER_NO_PERMISSION: *task_port* wasn't the kernel's task port. (Make sure **task_by_unix_pid()** is called by a process with superuser permission.)

KERN_LOADER_UNKNOWN_SERVER: *server_name* wasn't recognized.

SEE ALSO **kern_loader_add_server(), kern_loader_look_up()**

kern_loader_error(), kern_loader_error_string()

SUMMARY Display or return an error message

SYNOPSIS

#import <mach/mach.h>
#import <kernserv/kern_loader_error.h>

void **kern_loader_error**(const char **string*, kern_return_t *error*)
const char ***kern_loader_error_string**(kern_return_t *error*)

ARGUMENTS *string*: The string to be printed along with the error message.

error: The value returned by a Mach function.

DESCRIPTION These functions act like **mach_error()** and **mach_error_string()**, except that they also understand errors from the kernel-server loader functions.

The **kern_loader_error()** function prints to **stderr** the *string*, followed by the string corresponding to *error*, followed by *error* in parentheses. The **kern_loader_error_string()** function returns the string that corresponds to *error*.

EXAMPLE

```
error=kern_loader_delete_server(loader_port, kern_port, "mididriver");
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't delete mididriver", error);
    exit(3);
}
```

SEE ALSO **mach_error(), mach_error_string()**

kern_loader_get_log()

SUMMARY Request a message containing kernel log data

SYNOPSIS

```
#import <mach/mach.h>
#import <kernserv/kern_loader_types.h>
```

kern_return_t **kern_loader_get_log**(port_t *loader_port*, port_t *server_com_port*, port_t *reply_port*)

ARGUMENTS *loader_port*: **kern_loader** port, obtained from **kern_loader_look_up()**.

server_com_port: The loadable kernel server's communication port, obtained from **kern_loader_server_com_port()**.

reply_port: The port to which **kern_loader** should send the reply message.

DESCRIPTION This function requests a reply message containing data logged by a loadable kernel server. Before calling this function for the first time on a server, you should turn the server's logging on by calling **kern_loader_log_level()**.

You must supply the implementation of the reply message, as described in Chapter 3, "Using Loadable Kernel Servers."

Each item of logged data is preceded by a time stamp. The time stamp is a relative indicator of when the data was logged by the loadable kernel server.

EXAMPLE

```
r = kern_loader_look_up(&kl_port);
if (r != KERN_SUCCESS) {
    mach_error("Can't find kernel loader", r);
    exit(1);
}

r = port_allocate(task_self(), &reply_port);
if (r != KERN_SUCCESS) {
    mach_error("Can't allocate reply port", r);
    exit(1);
}
```

```
    /* Get the server's communication port. */
    r = task_by_unix_pid(task_self(), 0, &kern_port);
    if (r != KERN_SUCCESS) {
        mach_error("Error looking up kernel's port", r);
        exit(1);
    }
    r = kern_loader_server_com_port(kl_port, kern_port, MYDRIVER_NAME,
        &server_com_port);
    if (r != KERN_SUCCESS) {
        kern_loader_error("Error looking up server com port", r);
        exit(1);
    }

    /* Set the log level so we'll get log messages. */
    r = kern_loader_log_level(kl_port, server_com_port, LOG_NOTICE);
    if (r != KERN_SUCCESS) {
        kern_loader_error("Can't change log level", r);
        exit(1);
    }

    /* Get the first log message. */
    r = kern_loader_get_log(kl_port, server_com_port, reply_port);
    if (r != KERN_SUCCESS) {
        kern_loader_error("Error calling kern_loader_get_log", r);
        exit(1);
    }

    /* Listen for the asynchronous reply message. */
    listen(reply_port);
}

kern_loader_reply_t kern_loader_reply = {
    0,              /* argument to pass to function */
    0,              /* timeout for rpc return msg_send */
    0,              /* string function */
    0,              /* ping function */
    log_data        /* log_data function */
};

void listen(port_name_t port)
{
    char          msg_buf[kern_loader_replyMaxRequestSize];
    msg_header_t *msg = (msg_header_t *)msg_buf;
    kern_return_t r;
```

```
    while (1) {
        /* Receive the next message in the queue. */
        msg->msg_size = kern_loader_replyMaxRequestSize;
        msg->msg_local_port = port;
        r = msg_receive(msg, MSG_OPTION_NONE, 0);

        if (r != KERN_SUCCESS) {
            mach_error("listen msg_receive", r);
            exit(1);
        }

        /* Handle the message we just received. */
        kern_loader_reply_handler(msg, &kern_loader_reply);
    }
}

kern_return_t log_data(void *arg, printf_data_t log_data, unsigned int
log_data_count)
{
    kern_return_t r;

    /* Print the string we were passed, with our prefix. */
    printf("log_data: %s", log_data);

    /* Deallocate the memory used for the string. */
    vm_deallocate(task_self(), (vm_address_t)log_data,
        log_data_count*sizeof(*log_data));

    /* Get another log message. */
    r = kern_loader_get_log(kl_port, server_com_port, reply_port);
    if (r != KERN_SUCCESS) {
        kern_loader_error("Error calling kern_loader_get_log", r);
        exit(1);
    }
    return KERN_SUCCESS;
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_LOADER_UNKNOWN_SERVER: The server is either unknown or has been deallocated.

KERN_LOADER_SERVER_UNLOADED: The server is only allocated, not loaded.

KERN_LOADER_PORT_EXISTS: Someone is already receiving log messages for this server.

SEE ALSO **kern_loader_log_level(), kern_loader_look_up(), kern_loader_reply_handler(), kern_loader_server_com_port()**

---

kern_loader_load_server()

SUMMARY Load a loadable kernel server

SYNOPSIS

```
#import <mach/mach.h>
#import <kernserv/kern_loader.h>

kern_return_t kern_loader_load_server(port_t loader_port,
    server_name_t server_name)
```

ARGUMENTS *loader_port*: **kern_loader** port, obtained from **kern_loader_look_up()**.

server_name: The string associated with the server. For example, the MIDI driver's name is "mididriver".

DESCRIPTION This function loads a loadable kernel server that has already been allocated. If the server's relocatable object file has changed since allocation, then the server is allocated again from scratch. This function has no effect on servers that are already loaded; it simply returns KERN_SUCCESS.

EXAMPLE

```
/* Get kern_loader's port. */
error=kern_loader_look_up(&loader_port);
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't find kern_loader's port", error);
    exit(1);
}

/* Load the server. */
error=kern_loader_load_server(loader_port, "mididriver");
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't load the server", error);
    exit(2);
}
```

RETURN KERN_SUCCESS: The server was successfully loaded.

KERN_LOADER_UNKNOWN_SERVER: *server_name* wasn't recognized.

KERN_LOADER_SERVER_WONT_LOAD: The server couldn't be loaded.

SEE ALSO **kern_loader_look_up()**, **kern_loader_unload_server()**

---

kern_loader_log_level()

SUMMARY Set the level of data being logged by a loadable kernel server

SYNOPSIS

#import <mach/mach.h>
#import <kernserv/kern_loader.h>

kern_return_t **kern_loader_log_level**(port_t *loader_port*, port_t *server_com_port*, int *log_level*)

ARGUMENTS *loader_port*: **kern_loader** port, obtained from **kern_loader_look_up()**.

server_com_port: The loadable kernel server's communication port, obtained from **kern_loader_server_com_port()**.

log_level: An integer indicating the minimum priority of data to be logged. A value of zero turns logging off.

DESCRIPTION This function determines which data logged by a loadable kernel server gets kept. When the server is first loaded, none of its log messages are kept since its log level is initialized to zero. If you set *log_level* to a value greater than zero, then messages logged at a priority equal to or higher than *log_level* are kept. If you reset the log level to zero, no more log messages are kept until the log level is once again set to a positive value.

Each server can have its own conventions for log priorities.

EXAMPLE

```
r = kern_loader_look_up(&kl_port);
if (r != KERN_SUCCESS) {
    mach_error("Can't find kernel loader", r);
    exit(1);
}

/* Get the server's communication port. */
r = task_by_unix_pid(task_self(), 0, &kern_port);
if (r != KERN_SUCCESS) {
    mach_error("Error looking up kernel's port", r);
    exit(1);
}
r = kern_loader_server_com_port(kl_port, kern_port, MYDRIVER_NAME,
    &server_com_port);
if (r != KERN_SUCCESS) {
    kern_loader_error("Error looking up server com port", r);
    exit(1);
}

/* Set the log level so we'll get log messages. */
r = kern_loader_log_level(kl_port, server_com_port, LOG_NOTICE);
if (r != KERN_SUCCESS) {
    kern_loader_error("Can't change log level", r);
    exit(1);
}
```

RETURN KERN_SUCCESS: The log level was successfully set.

KERN_LOADER_UNKNOWN_SERVER: *server_com_port* wasn't valid.

KERN_LOADER_SERVER_UNLOADED: The server isn't currently loaded.

SEE ALSO **kern_loader_look_up(), kern_loader_server_com_port(), kern_loader_log_level()**

kern_loader_look_up()

SUMMARY Get the **kern_loader** port

SYNOPSIS

#import <mach/mach.h>
#import <kernserv/kern_loader_types.h>

kern_return_t **kern_loader_look_up**(port_t **loader_port*)

ARGUMENTS *loader_port*: Returns the port on which **kern_loader** receives messages.

DESCRIPTION This function returns the service port for the kernel-server loader.

EXAMPLE

```
/* Get kern_loader's port. */
error=kern_loader_look_up(&loader_port);
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't find kern_loader's port", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: The call succeeded.

SEE ALSO **kern_loader_server_com_port()**

kern_loader_ping()

SUMMARY Request a synchronization message

SYNOPSIS

#import <mach/mach.h>
#import <kernserv/kern_loader_types.h>

kern_return_t **kern_loader_ping**(port_t *loader_port*, port_t *ping_port*, int *id*)

ARGUMENTS *loader_port*: **kern_loader** port, obtained from calling **kern_loader_look_up()**.

ping_port: The port to which the ping message should be sent.

id: A value to be sent in the message. You can use this as you wish.

DESCRIPTION You can use this function to make sure that all outstanding status messages have been sent to your program. For example, if you call **kern_loader_status_port()**, you might want to call **kern_loader_ping()** at some point afterward. When you receive the ping message, you'll know that you've received all status messages that were queued before you called **kern_loader_ping()**.

Another reason to call **kern_loader_ping()** is to check whether **kern_loader** has fallen into an unresponsive state.

You must implement the ping message yourself, as described in Chapter 3. The **kern_loader_ping()** function returns a value indicating whether the message was successfully sent to *ping_port*.

EXAMPLE

```
    r = kern_loader_look_up(&kl_port);
    if (r != KERN_SUCCESS) {
        mach_error("kl_util: can't find kernel loader", r);
        exit(1);
    }

    r = port_allocate(task_self(), &reply_port);
    if (r != KERN_SUCCESS) {
        mach_error("kl_util: can't allocate reply port", r);
        exit(1);
    }

    /* Create a thread to listen on reply_port. */
    cthread_detach(cthread_fork((cthread_fn_t)ping_thread,
        (any_t)reply_port));

    /* . . . */

    /* Get a ping message sent to the reply port. */
    r=kern_loader_ping(kl_port, reply_port, 0);

    /* Wait for ping() to kill us.  Exit if we receive a signal. */
    pause();
    exit(0);
}
```

```
kern_loader_reply_t kern_loader_reply = {
    0,        /* argument to pass to function */
    0,        /* timeout for rpc return msg_send */
    0,        /* string function */
    ping,     /* ping function */
    0         /* log_data function */
};

void ping_thread(port_name_t port)
{
    char          msg_buf[kern_loader_replyMaxRequestSize];
    msg_header_t *msg = (msg_header_t *)msg_buf;
    kern_return_t r;

    /* message handling loop */
    while (TRUE) {
        /* Receive the next message in the queue. */
        msg->msg_size = kern_loader_replyMaxRequestSize;
        msg->msg_local_port = port;
        r = msg_receive(msg, MSG_OPTION_NONE, 0);
        if (r != KERN_SUCCESS)
            break;

        /* Handle the message we just received. */
        kern_loader_reply_handler(msg, &kern_loader_reply);
    }

    /* We get here only if msg_receive returned an error. */
    mach_error("ping_thread receive", r);
    exit(1);
}

/* This function is called after a kern_loader_ping. */
kern_return_t ping (void *arg, int id)
{
    exit(0);    /* Kill this process. */
}
```

SEE ALSO **kern_loader_reply_handler()**

kern_loader_reply_handler()

SUMMARY Handle a message from the kernel-server loader

SYNOPSIS

#import <mach/mach.h>
#import <kernserv/kern_loader_reply_handler.h>

kern_return_t **kern_loader_reply_handler**(msg_header_t **msg*,
kern_loader_reply_t **kern_loader_reply*)

ARGUMENTS

msg: The message you just received from the kernel-server loader.

kern_loader_reply: A pointer to the structure that specifies which of your functions handle each type of reply from the kernel-server loader.

DESCRIPTION You must use this function if you use **kern_loader_ping()**, **kern_loader_get_log()**, or **kern_loader_status_port()**. Those routines cause an asynchronous reply message from the kernel-server loader; this reply message must be passed to **kern_loader_reply_handler()**, which forwards the message data to your *ping_func()*, *log_func()*, or *string_func()* function.

This function returns the value that is returned by your *ping_func()*, *log_func()*, or *string_func()* function. See Chapter 3 for more information on implementing these functions.

EXAMPLE

```
kern_loader_reply_t kern_loader_reply = {
    0,              /* argument to pass to function */
    0,              /* timeout for rpc return msg_send */
    0,              /* string function */
    0,              /* ping function */
    log_data        /* log_data function */
};

void listen(port_name_t port)
{
    char            msg_buf[kern_loader_replyMaxRequestSize];
   msg_header_t *msg = (msg_header_t *)msg_buf;
    kern_return_t r;
```

```
    while (1) {
        /* Receive the next message in the queue. */
        msg->msg_size = kern_loader_replyMaxRequestSize;
        msg->msg_local_port = port;
        r = msg_receive(msg, MSG_OPTION_NONE, 0);

        if (r != KERN_SUCCESS) {
            mach_error("listen msg_receive", r);
            exit(1);
        }

        /* Handle the message we just received. */
        kern_loader_reply_handler(msg, &kern_loader_reply);
    }
}
```

SEE ALSO **kern_loader_get_log(), kern_loader_ping(), kern_loader_status_port()**

---

kern_loader_server_com_port()

SUMMARY Get a loadable kernel server's communication port

SYNOPSIS

#import <mach/mach.h>
#import <kernserv/kern_loader.h>

kern_return_t **kern_loader_server_com_port**(port_t *loader_port*, port_t *task_port*, server_name_t *server_name*, port_t **server_com_port*)

ARGUMENTS

loader_port: **kern_loader** port, obtained from calling **kern_loader_look_up()**.

task_port: The kernel port for the task in which the loadable kernel server is executing. This is returned by **kern_loader_server_task_port()**.

server_name: The string associated with the server. For example, the name of the MIDI driver is "mididriver".

server_com_port: Returns the server's communication port.

DESCRIPTION A loadable kernel server's communication port is used for logging-related functions, such as **kern_loader_get_log()** and **kern_loader_log_level()**.

EXAMPLE

```
/* Get kern_loader's port. */
r = kern_loader_look_up(&kl_port);
if (r != KERN_SUCCESS) {
    mach_error("Can't find kernel loader", r);
    exit(1);
}

/* Get the kernel's task port. */
r = task_by_unix_pid(task_self(), 0, &kern_port);
if (r != KERN_SUCCESS) {
    mach_error("Error looking up kernel's port", r);
    exit(1);
}

/* Get the server's com port. */
r = kern_loader_server_com_port(kl_port, kern_port, MYDRIVER_NAME,
    &server_com_port);
if (r != KERN_SUCCESS) {
    kern_loader_error("Error looking up server com port", r);
    exit(1);
}
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_LOADER_NO_PERMISSION: *task_port* wasn't the server's task port.

KERN_LOADER_UNKNOWN_SERVER: *server_name* wasn't recognized.

SEE ALSO **kern_loader_get_log(), kern_loader_log_level(), kern_loader_look_up()**

kern_loader_server_info()

SUMMARY Get information about a loadable kernel server

SYNOPSIS

#import <mach/mach.h>
#import <kernserv/kern_loader_reply.h>

kern_return_t **kern_loader_server_info**(port_t *loader_port*, port_t *task_port*,
server_name_t *server_name*, server_state_t **server_state*,
vm_address_t **load_address*, vm_size_t **load_size*, server_reloc_t *relocatable*,
server_reloc_t *loadable*, port_name_array_t **port_list*, unsigned int **port_list_count*,
port_name_string_array_t **port_names*, unsigned int **port_names_count*,
boolean_array_t **advertised*, unsigned int **advertised_count*)

ARGUMENTS *loader_port*: **kern_loader** port, obtained from calling **kern_loader_look_up()**.

task_port: The kernel's task port, obtained using **task_by_unix_pid()**. Specify PORT_NULL if you don't want to have data returned in *port_list*.

server_name: The string associated with the server. For example, the name of the MIDI driver is "mididriver".

server_state: Returns the state of the loadable kernel server. The value is one of the following: Zombie, Allocating, Allocated, Loading, Loaded, Unloading, Deallocated (as defined in the header file **kernserv/kern_loader_types.h**).

load_address: Returns the address in the kernel address space where the server starts.

load_size: Returns the number of bytes used by the server. *load_address* + *load_size* – 1 is the last address in the kernel map that's used by the server's text and data.

relocatable: Returns the location of the relocatable object file for the server.

loadable: Returns the location of the loadable object file (if any) for this server. This is a file created by **kern_loader** from *relocatable* and then loaded against **/mach**.

port_list: Returns the ports that the server has made available to **kern_loader**, using the HMAP or SMAP load command. If you don't pass in the correct *task_port*, this list will consist of null ports.

port_list_count: Returns the number of ports that the server has made available to **kern_loader**. Even if *task_port* isn't valid, and nothing is returned in *port_list*, this argument holds the number of ports that would have been returned.

port_names: Returns the strings associated with the ports in *port_list*.

port_names_count: Returns the number of names in *port_names*. This number is the same as *port_list_count*.

advertised: For each entry in *port_list*, returns true if the port is advertised with the Network Name Server.

advertised_count: Returns the number of entries in *advertised*. This number is the same as *port_list_count*.

DESCRIPTION **kern_loader_server_info()** returns information about a particular loadable kernel server.

EXAMPLE

```
/* Get kern_loader's port. */
error=kern_loader_look_up(&loader_port);
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't find kern_loader's port", error);
    exit(1);
}

/* Get the information. */
error=kern_loader_server_info(loader_port, PORT_NULL, "mididriver",
    &server_state, &load_addr, &load_size, relocatable, loadable,
    (port_name_array_t *)&scratch, &count, &port_names, &count,
    &advertised, &count);
if (error != KERN_SUCCESS)
    kern_loader_error("Couldn't get info on mididriver", error);
else
    printf("The relocatable object file is located at:  %s\n",
    relocatable);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_LOADER_UNKNOWN_SERVER: *server_name* wasn't recognized.

SEE ALSO **kern_loader_look_up(), kern_loader_server_list()**

kern_loader_server_list()

SUMMARY Get the names of all known loadable kernel servers

SYNOPSIS

#import <mach/mach.h>
#import <kernserv/kern_loader.h>

kern_return_t **kern_loader_server_list**(port_t *loader_port*,
server_name_array_t **server_names*, unsigned int **server_names_count*)

ARGUMENTS *loader_port*: **kern_loader** port, obtained from calling **kern_loader_look_up()**.

server_names: Returns an array whose entries are the strings associated with all known servers.

server_names_count: Returns the number of entries in *server_names*.

DESCRIPTION Use this function to get the string associated with each loadable kernel server that **kern_loader** is keeping track of.

EXAMPLE

```
r = kern_loader_look_up(&loader_port);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't get loader_port", r);
    exit(1);
}

r = kern_loader_server_list(loader_port, &server_names, &count);
if (r != KERN_SUCCESS)
    kern_loader_error("Couldn't get the list", r);
else
    for (i=0; i<count; i++)
        printf("Server %d:  %s\n", i, server_names[i]);
```

RETURN KERN_SUCCESS: The call succeeded.

SEE ALSO **kern_loader_look_up()**, **kern_loader_server_info()**

kern_loader_server_task_port()

SUMMARY Get the task port of a loadable kernel server

SYNOPSIS

#import <mach/mach.h>
#import <kernserv/kern_loader.h>

kern_return_t **kern_loader_server_task_port**(port_t *loader_port*, port_t *kernel_port*, server_name_t *server_name*, port_t **server_task_port*)

ARGUMENTS *loader_port*: **kern_loader** port, obtained from calling **kern_loader_look_up()**.

kernel_port: The kernel's task port.

server_name: The string associated with a loaded server. For example, the name of the MIDI driver is "mididriver".

server_task_port: Returns the kernel port for the task in which the loadable kernel server is executing.

DESCRIPTION This function returns the task port of the server. Each loadable kernel server currently executes in its own task, but uses the kernel address space. The port returned by **kern_loader_server_task_port()** isn't necessary for any other kernel-server loader functions, but might be useful for gathering debugging information.

EXAMPLE

```
port_t              loader_port, kernel_task, server_port;
kern_return_t       r;
port_name_array_t   names;
unsigned int        i, names_count, types_count;
port_type_array_t   types;

r = kern_loader_look_up(&loader_port);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't get loader_port", r);
    exit(1);
}

r = task_by_unix_pid(task_self(), 0, &kernel_task);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't get kernel_task", r);
    exit(2);
}
```

```
r = kern_loader_server_task_port(loader_port, kernel_task, "mididriver",
    &server_port);
if (r != KERN_SUCCESS)
    kern_loader_error("Couldn't get the server port", r);
else
    printf("Midi's task port is %d\n", server_port);

r = port_names((task_t)server_port, &names, &names_count, &types,
    &types_count);
if (r != KERN_SUCCESS)
    mach_error("Error calling port_names()", r);
else
    for (i=0; i<names_count; i++)
        printf("Port %d has type %d\n", names[i], types[i]);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_LOADER_NO_PERMISSION: *task_port* wasn't the server's task port.

KERN_LOADER_UNKNOWN_SERVER: *server_name* wasn't recognized.

SEE ALSO **kern_loader_look_up(), kern_loader_server_com_port()**

---

kern_loader_status_port()

SUMMARY Specify a port for **kern_loader** to send status to

SYNOPSIS **#import <mach/mach.h>**
#import <kernserv/kern_loader.h>

kern_return_t **kern_loader_status_port**(port_t *loader_port*, port_t *listen_port*)

ARGUMENTS *loader_port*: **kern_loader** port, obtained from calling **kern_loader_look_up()**.

listen_port: The port we want to receive the status messages on.

DESCRIPTION Use this function to get general status from **kern_loader**. You can receive many reply messages as the result of just one call to **kern_loader_status_port()**.

You must define the function that handles status reply messages, as described in Chapter 3. This function receives the status string along with its priority, using the priorities defined in the header file **sys/syslog.h** (LOG_EMERG, LOG_ALERT, and so on).

EXAMPLE

```
    r = kern_loader_look_up(&kl_port);
    if (r != KERN_SUCCESS) {
        mach_error("kl_util: can't find kernel loader", r);
        exit(1);
    }

    r = port_allocate(task_self(), &status_port);
    if (r != KERN_SUCCESS) {
        mach_error("kl_util: can't allocate reply port", r);
        exit(1);
    }

    /* Get generic status messages on this port. */
    r = kern_loader_status_port(kl_port, status_port);
    if (r != KERN_SUCCESS) {
        kern_loader_error("Couldn't specify status port", r);
            exit(1);
    }

    /* Create a thread to listen on status_port. */
    cthread_detach(cthread_fork((cthread_fn_t)receive_thread,
        (any_t)status_port));

    /*
     * Sleep for a while so we can enter kl_util commands at a shell
     * window.  The output of all commands (except status lines from
     * kl_util -s) will show up in both the window that's running this
     * program and in the window that's running kl_util.  (kl_util
     * also has a status port registered.)
     */
    sleep(30);

    exit(0);
}
```

```
kern_loader_reply_t kern_loader_reply = {
    0,              /* argument to pass to function */
    0,              /* timeout for rpc return msg_send */
    print_string,   /* string function */
    0,              /* reply_ping function */
    0               /* log_data function */
};

void receive_thread(port_name_t port)
{
    char          msg_buf[kern_loader_replyMaxRequestSize];
    msg_header_t *msg = (msg_header_t *)msg_buf;
    kern_return_t r;

    /* message handling loop */
    while (TRUE) {
        /* Receive the next message in the queue. */
        msg->msg_size = kern_loader_replyMaxRequestSize;
        msg->msg_local_port = port;
        r = msg_receive(msg, MSG_OPTION_NONE, 0);
        if (r != KERN_SUCCESS)
            break;

        /* Handle the message we just received. */
        kern_loader_reply_handler(msg, &kern_loader_reply);
    }

    /* We get here only if msg_receive returned an error. */
    mach_error("receive_thread receive", r);
    exit(1);
}

/* Called every time kern_loader has status to report. */
kern_return_t print_string(void *arg, printf_data_t string,
    u_int string_count, int level)
{
    /* If the string is empty, return. */
    if (string_count == 0 || !string)
        return KERN_SUCCESS;

    /* Print the string we were passed, with our prefix. */
    printf("print_string: %s", string);

    return KERN_SUCCESS;
}
```

RETURN KERN_SUCCESS: The call succeeded.

SEND_INVALID_PORT: *listen_port* isn't a valid port.

SEE ALSO **kern_loader_look_up(), kern_loader_reply_handler()**

---

kern_loader_unload_server()

SUMMARY Unload a loadable kernel server

SYNOPSIS

```
#import <mach/mach.h>
#import <kernserv/kern_loader.h>
```

kern_return_t **kern_loader_unload_server**(port_t *loader_port*, port_t *task_port*, server_name_t *server_name*)

ARGUMENTS *loader_port*: **kern_loader** port, obtained from calling **kern_loader_look_up()**.

task_port: The kernel's task port, obtained using **task_by_unix_pid()**.

server_name: The string associated with the server. For example, the name of the MIDI driver is "mididriver".

DESCRIPTION Use this function to unload a running loadable kernel server, leaving it allocated.

EXAMPLE

```
r = kern_loader_look_up(&loader_port);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't get loader_port", r);
    exit(1);
}

r = task_by_unix_pid(task_self(), 0, &kernel_task);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't get kernel_task", r);
    exit(2);
}

r = kern_loader_unload_server(loader_port, kernel_task,
    "NextDimension");
if (r != KERN_SUCCESS)
    kern_loader_error("Couldn't unload the server", r);
```

RETURN KERN_SUCCESS: The server was successfully unloaded.

KERN_LOADER_SERVER_UNLOADED: The server was already unloaded.

KERN_LOADER_NO_PERMISSION: *task_port* wasn't the kernel's task port. (Make sure **task_by_unix_pid()** is called by a process with superuser permission.)

KERN_LOADER_UNKNOWN_SERVER: *server_name* wasn't recognized.

SEE ALSO **kern_loader_load_server()**, **kern_loader_look_up()**

Part 2
Writing Loadable Kernel Servers

Contents of Part 2

5-1 Chapter 5: Overview of Loadable Kernel Servers
5-1 Advantages and Disadvantages of Loadable Kernel Servers
5-2 Supported Types of Loadable Kernel Servers
5-3 How Loadable Kernel Servers Work
5-5 Before You Start
5-6 Hardware You'll Need
5-6 Documentation You'll Need

6-1 Chapter 6: Designing Loadable Kernel Servers
6-1 Choosing an Interface for Your Server
6-2 Tips for Writing Server Code
6-3 Functions Your Server Can Call
6-3 Executing as the Result of an Interrupt
6-4 Building In Debugging Code
6-4 Displaying Debugging Information
6-4 Using **kern_serv_log()**
6-5 Using **printf()**
6-5 Checking Assumptions
6-5 Using **ASSERT()**
6-5 Using **probe_rb()**
6-6 Kernel-Server Loader Requirements
6-6 Instance Variable
6-7 Writing Functions for **kern_loader** to Call
6-7 Considerations for Message-Based Servers
6-8 Server versus Handler Interfaces
6-10 Sending and Receiving Out-of-Line Data
6-13 Considerations for UNIX-Based Servers

6-13 UNIX Device Entry Points
6-13 Character-Device Entry Points
6-16 Block-Device Entry Points
6-17 Inserting UNIX Servers into Device Switch Tables

7-1 Chapter 7: NeXTbus Device Drivers
7-2 NeXTbus Address Space
7-4 Writing Code to Control the Hardware
7-4 Using BusProbe
7-4 Mapping from Physical to Virtual Memory Addresses
7-5 Transferring Data
7-6 NeXTbus Byte Ordering
7-6 Writing an Interrupt Handler
7-8 NeXTbus Board Registers
7-9 CPU Board NBIC Registers
7-9 NBIC Control Register
7-10 Other NeXTbus Board Registers
7-10 Identification Bytes (NBIC ID Register)
7-13 Interrupt Byte
7-13 Interrupt Mask Byte

8-1 Chapter 8: Network Modules
8-2 Overview
8-3 Network Objects
8-3 Network Buffers (Netbufs)
8-3 Network Interfaces (Netifs)
8-4 Functions Implemented in the Network Module
8-4 Callback Function
8-6 Initialization Function
8-6 Input Function
8-8 Output Function
8-10 Getbuf Function
8-11 Control Function
8-12 Notes for Specific Interfaces
8-12 Ethernet Interfaces
8-13 TCP/IP Interfaces

9-1 Chapter 9: Building, Loading, and Debugging Loadable Kernel Servers
9-1 Building a Loadable Kernel Server
9-3 Loading a Server into the Kernel
9-4 Tools for Debugging Servers

9-5 NMI Mini-Monitor Window
9-5 ROM Monitor Window
9-5 Panic Window
9-6 System Console
9-6 Other Debugging Tools
9-7 Debugging Servers with GDB
9-7 Setting Up the Computers
9-8 Setting Up the Files
9-9 Files Needed by the Slave Computer
9-9 Files Needed by the Master Computer
9-9 Starting Up GDB
9-11 Debugging with GDB
9-12 Ending the Debugging Session

10-1 Chapter 10: Kernel Support Functions
10-2 General Functions
10-54 Network Functions

A-1 Appendix A: Utilities for Loadable Kernel Servers
A-2 Kernel-Server Loader (**kern_loader**)
A-2 Starting **kern_loader**
A-4 Kernel-Server Linker (**kl_ld**)
A-5 Load Commands
A-7 Unload Commands
A-8 Kernel-Server Utility (**kl_util**)
A-10 Kernel-Server Log Command (**kl_log**)

B-1 Appendix B: The ROM Monitor and NMI Mini-Monitor
B-1 ROM Monitor Commands
B-2 Open Address Register
B-2 Open Data Register
B-3 Open Processor Register
B-3 Open System Register
B-4 Examine Memory Locations
B-5 Open Function Code
B-5 Set Input Radix
B-6 NMI Mini-Monitor Commands
B-6 Set or Examine Any Kernel Flag
B-6 Set the IP Address

C-1 Appendix C: Summary of Kernel Support Functions
C-1 General Functions
C-2 Time Functions
C-2 Memory Functions
C-3 Critical Section and Synchronization Functions
C-3 General Task and Thread Functions
C-4 Port and Message Functions
C-4 Hardware Interface Functions
C-5 Logging and Debugging Functions
C-5 UNIX Support Functions
C-6 Miscellaneous Functions
C-6 Network Functions
C-6 Netif Functions
C-8 Netbuf Functions
C-8 Miscellaneous Functions

Introduction to Part 2

The following chapters describe how to write loadable kernel servers for NeXT computers.

- Chapter 5, "Overview of Loadable Kernel Servers," describes loadable kernel servers, with the aim of helping you decide whether you need to write one.
- Chapter 6, "Designing Loadable Kernel Servers," provides the basic information you'll need to design a loadable kernel server.
- Chapter 7, "NeXTbus Device Drivers," gives information that's necessary only for writing NeXTbus device drivers.
- Chapter 8, "Network Modules," gives information that's necessary only if you're writing a server that fits into the networking system.
- Chapter 9, "Building, Loading, and Debugging Loadable Kernel Servers," gives information on using tools to help test your server. It also provides the information you need to build your server for final release.
- Chapter 10, "Kernel Support Functions," gives detailed descriptions of the functions that are available only to loadable kernel servers.

In addition, there are three appendices:

- Appendix A, "Utilities for Loadable Kernel Servers," describes the utilities that support loadable kernel servers.
- Appendix B, "The ROM Monitor and NMI Mini-Monitor," describes ROM monitor and NMI mini-monitor commands that are of interest to only a few developers. Other, more widely used commands are described in Chapter 9 and in the *NeXTSTEP Network and System Administration* manual.

- Appendix C, "Summary of Kernel Support Functions," summarizes the functions described in Chapter 10.

Some topics aren't covered here in detail; instead, you're referred to a generally available book on the subject, or to an on-line source of the information. For example, Egan and Teixeira's *Writing a UNIX Device Driver* describes writing UNIX-style drivers and gives general driver-writing tips; those subjects aren't described in detail in this manual.

5 *Overview of Loadable Kernel Servers*

This chapter should help you decide whether you need to write a loadable kernel server, as well as give you the background information you need to begin writing one. A *loadable kernel server* is any code that's added to the NeXT™ Mach kernel after the system has been booted. Loadable kernel servers are the only way to make third-party code execute within the kernel, as opposed to executing at *user level.*

This chapter first discusses the advantages and disadvantages of loadable kernel servers, and then describes the types of loadable kernel servers that you can write. So that you don't unnecessarily write a loadable kernel server, a list of existing NeXT device drivers is presented. Next, this chapter describes in general how loadable kernel servers work. Finally, it describes the hardware and documentation that you'll need if you write a server.

Advantages and Disadvantages of Loadable Kernel Servers

Most of the code that runs on a NeXT computer executes at user level. For example, all NeXTSTEP™ applications, such as Edit and Digital Librarian™, execute at user level, although they occasionally request services from the kernel. Command-line utilities, such as **make**, also execute at user level. Even system-wide resources such as the Window Server, with its Display PostScript® interpreter, run at user level. User-level programs have several advantages over kernel code:

- Debugging them is much easier than debugging kernel code. For example, running a debugger on kernel code requires two computers and frequently freezes the operating system of the computer that's being debugged.

- Errors in user-level code, unlike those in kernel code, don't cause the computer to *panic* (a state in which the operating system stops running; it can cause data loss and can be corrected only by rebooting).

- User-level programs usually don't decrease the overall performance of the system as much as if they were executed in the kernel. For example, code executed in the kernel causes the kernel to reserve more physical memory, reducing the memory that can be used for user-level programs.

However, sometimes it's necessary to execute code in the kernel, especially when:

- User-level code just can't do the job. For example, the NeXT networking architecture currently requires that network protocols be executed inside the kernel.

- User-level code can't do the job quickly enough. For example, if you have a NeXTbus™ device, you can either write a user-level program that calls the slot driver (a driver provided in the NeXTbus Development Kit, NeXT product number N7002) or write your own loadable kernel server to control the device. The performance requirements of the device determine whether it's necessary to write your own loadable kernel server. The NeXTbus interface bus is discussed in Chapter 7, "NeXTbus Device Drivers."

Supported Types of Loadable Kernel Servers

Currently, third parties can't write device drivers for non-NeXTbus interfaces, such as SCSI and serial ports. Third parties can write three types of loadable kernel servers:

- Device drivers for NeXTbus boards
- Network protocols
- Network packet sniffers

NeXT computers have several drivers that take care of most interface needs:

Driver	Interface
audio	Sound input and output
fd	Internal floppy disk drive
MIDI	Serial interfaces to MIDI devices
np	NeXT laser printer
od	Internal optical disk
sd	SCSI disk drives of all kinds, including hard disks, floppy disks, and CD-ROMs
sg	Concurrent access to up to four SCSI devices of any kind
slot	NeXTbus devices (this driver is provided in the NeXTbus Development Kit)
sound/DSP	DSP56001 digital signal processor
st	SCSI tape drives
zs	Serial port devices

The fd, np, od, sd, sg, st, and zs drivers are described in the UNIX manual pages, which are available through Digital Librarian. The MIDI and sound/DSP drivers are described in the *NeXTSTEP General Reference*, which is also available through Digital Librarian. The audio driver is accessible only indirectly, through Sound Kit™ objects; these objects are described in the *General Reference*.

How Loadable Kernel Servers Work

One of the main features of loadable kernel servers is that they can be loaded into the kernel at any time after the computer has booted. This has several benefits. One is that you can add a server without recompiling the kernel. Another is that a server can be unloaded when it's no longer required, saving memory and other resources. You can also unload and reload a server, which speeds up development because you don't have to reboot every time you change the server.

The *kernel-server loader* (**kern_loader**) is the task that loads and unloads loadable kernel servers. You can communicate with **kern_loader** either with the *kernel-server utility* (**kl_util**, which is described later), or by using the kernel-server loader functions, a group of user-level functions that are described in Chapter 3, "Using Loadable Kernel Servers."

Loadable kernel servers have three states:

- Allocated—**kern_loader** has allocated space and resources for the loadable kernel server and is listening for messages to its ports. However, the server isn't currently running.
- Loaded—The loadable kernel server is running. It runs as a thread in a non-kernel task, but with the kernel's address map.
- Unallocated—**kern_loader** has no space or other resources allocated for the loadable kernel server.

Loadable kernel servers can be built with or without Mach messages. A server that's based on messages has the advantage that it's easily loaded at the moment it's needed. This works because **kern_loader**, when it allocates resources for a server, creates the server's ports and advertises them with the Network Name Server. When a message is received on one of these advertised ports, **kern_loader** loads the allocated server into the kernel and forwards the advertised ports along with the received message to the now loaded and running server.

Loadable kernel servers that don't use messages must be loaded into the system as soon as they're allocated in **kern_loader**. Like other loadable kernel servers, though, they can be allocated at any time and unloaded when they're no longer needed. Since newly loaded servers aren't known to the kernel, those that aren't message-based must add their entry points to the kernel. For example, a UNIX-style server (a server accessed with UNIX system calls) must insert pointers to its entry points into device switch tables.

A loadable kernel server doesn't have to use messages if it's accessed only by table lookups. For example, UNIX-style servers and networking protocols can be implemented without messages.

When started, **kern_loader** reads a configuration file (**/etc/kern_loader.conf**) and allocates the listed kernel servers. (Servers can also be allocated later, using **kl_util** or the kernel-server loader functions.) During allocation, **kern_loader** extracts the following information from the server's object file:

- The name of the loadable kernel server
- The names of its ports, and whether they should be advertised with the Network Name Server
- Which function should be called when a message is received on a particular port

- Whether the server should be loaded into the kernel immediately or wait for a received message
- Whether the server should be *wired down* (made memory-resident in kernel virtual memory)
- Initialization functions to call when the server is loaded
- Shutdown functions to call when the server is unloaded

You specify some or all the information listed above to command scripts that are read by the kernel-server linker, **kl_ld**. Chapter 9, "Building, Loading, and Debugging Loadable Kernel Servers," discusses how to compile and link loadable kernel servers; Appendix A, "Utilities for Loadable Kernel Servers," gives details on **kl_ld** and the format of command scripts.

When **kern_loader** allocates a server, it allocates memory within the Mach kernel and relocates the file against the running kernel's symbol table at the allocated address. The resulting code and data are directly copied into the kernel virtual address space when the server is loaded. While you're debugging your server, you need to specify that **kern_loader** should also save the relocated code and data into a *loadable object file*, which is used by the debugger. (The original object file—the one that **kern_loader** uses to create the loadable object file—is called the *relocatable object file*.)

During the loading process, **kern_loader** not only loads the server but also initializes it. Initialization includes associating ports with functions and calling any initialization functions. The server is also wired down if requested.

Before **kern_loader** deallocates a server, it calls the server's shutdown functions. The server is responsible for making sure that it has freed all the kernel resources that it requested (such as dynamically allocated memory).

Before You Start

To write a loadable kernel server, you must be able to program in C. Depending on the kind of loadable kernel server you're writing, you probably don't need to be familiar with UNIX® internals. However, a general background in writing interrupt-driven device drivers or networking protocols is useful.

If you've determined that you need to write a loadable kernel server, you'll need to gather some equipment and documentation.

Hardware You'll Need

To write a server, you'll need the following equipment:

- Two NeXT computers that are running the same major release of NeXTSTEP (for example, NeXTSTEP Release 3).
- A working network connection for both computers.
- A NeXTbus Interface Chip (NBIC) on the CPU board of the test computer (only if you're writing a driver for a NeXTbus device). The NBIC is already installed in every NeXTcube™ computer and 68040 Upgrade Board.
- The hardware (if any) your server will control, installed in or connected to one of the computers.

See Chapter 9 for more information on setting up the two computers.

Documentation You'll Need

Depending on the type of server you're writing, you'll need some or all of the documentation listed in this section. You'll also need the server examples that are under **/NextLibrary/Documentation/NextDev/Examples/ServerVsHandler**.

Part 1 of this manual, "The Mach Operating System," is useful for its description of the NeXT Mach operating system and of Mach messaging, as well as documentation of Mach functions.

You should also have *NeXTSTEP Development Tools*, which is part of the *NeXTSTEP Developer's Library.* It describes the compiler and debugger that you'll use.

You might need to have the *NeXTSTEP Network and System Administration* manual on hand, especially when you're setting up your computers.

If you're writing a driver for a NeXTbus board, you'll need the hardware specifications and installation guide. These are all part of the NeXTbus Development Kit.

- *NeXTbus Interface Chip Specification*
- *NeXTbus Specification*
- *NeXTbus Development Kit Installation Guide*

The following book has good information for any driver writer. It also has specific information on UNIX drivers.

Writing a UNIX Device Driver. Janet I. Egan and Thomas J. Teixeira. John Wiley & Sons, Inc., 1988.

If you're writing a server with a UNIX-style interface, you might need the information in the following book.

The Design and Implementation of the 4.3BSD UNIX Operating System. Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. Addison-Wesley, 1989.

6 *Designing Loadable Kernel Servers*

This chapter provides the basic information required to design loadable kernel servers. It describes the code that every server must have to be integrated into the kernel and to provide services to its clients. It also discusses the functions that loadable kernel servers can call, and it gives tips to help you write correct code that makes debugging easier.

Besides this chapter, you should also read either Chapter 7, "NeXTbus Device Drivers," or Chapter 8, "Network Modules," for detailed information about the type of server you're writing.

Choosing an Interface for Your Server

Before you can start writing your server, you have to decide whether it needs a message-based interface or a UNIX-style interface. Network modules such as network protocols primarily use a third kind of interface (called a *network interface*, which is described in Chapter 8). However most network modules have additional interface functions that are provided through either a message-based or a UNIX-style interface.

It is recommended that you use the Mach Interface Generator (MiG) to write a message-based interface. One advantage of using a message-based interface is that your server can stay unloaded until the moment it's needed. Message-based servers, besides having more intuitive interfaces than UNIX-style servers, also have the advantage of MiG's network independence. For example, a graphics device with a message-based server could easily be accessed from any computer on the network. MiG and Mach messages are described in detail in Chapter 2, "Using Mach Messages."

Unfortunately, some servers can't use message-based interfaces. For example, servers that interact with the UNIX file system—such as disk drivers—need to supply a UNIX-style interface with UNIX entry points (*xxx*_**open()**, *xxx*_**close()**, and so on).

Tips for Writing Server Code

When developing your server, you should start by designing your server's interface. When you're ready to start writing code, use a skeleton or sample server as the framework, and then add functionality a little at a time. You might want to put all the major interface functions in place, but just have each one print a message that says it's been called. This approach will make debugging your server much easier than if you implement large amounts of code at once.

As you write the code for your server, keep the following in mind:

- Using **register** isn't necessary. It's easier and often better to let the compiler decide what to put into registers.
- You must use **volatile** for variables that refer to hardware addresses, or that can be modified by interrupt functions or other threads.
- Beware of hardware registers that have side effects when accessed, or that contain different information when you read them than when you write to them.
- Don't declare large variables in functions. Instead, if you need a large variable, declare a pointer to it and then dynamically allocate space with **kalloc()** or **kget()**. (Automatic variables are allocated on the kernel stack. Since the kernel stack is only 4 KB, large variables can easily cause stack overflow and result in system panics that aren't easily debugged.)
- Don't recursively call functions. (Like large automatic variables, recursion can cause stack overflow.)

Functions Your Server Can Call

Your loadable kernel server can use the functions described in Chapter 10, "Kernel Support Functions." In addition, it can use almost all Mach kernel functions, which are listed in Chapter 4, "Mach Functions."

However, loadable kernel servers can't use functions that are not defined in the Mach kernel. For example, a few Mach kernel functions don't work because they rely on data that's only available at user level. For example, **mach_error()** can't be used in loadable kernel servers because it prints to **stderr**, which the Mach kernel doesn't have access to.

Note: You can't use C Thread, Network Name Server, Bootstrap Server, or Kernel-Server Loader functions or macros in your server, because they aren't part of the Mach kernel.

Warning: Loadable kernel servers run outside of the kernel task, even though they use the kernel address map. Be sure to specify the correct task and map to Mach kernel functions.

Executing as the Result of an Interrupt

When a function in your server is called as the direct result of an interrupt, the computer stops all normal processing until the function exits. Even if your server doesn't handle hardware interrupts, it can run as the result of a software interrupt. Specifically, functions that run because they were scheduled by a call to **ns_timeout()** or **ns_abstimeout()** (as described in Chapter 10) are running as the result of a software interrupt.

A server that's executing as the result of an interrupt must take certain precautions:

- It must not sleep.
- It must not call any functions that might sleep.
- If it needs to allocate memory, it should use **kget()** (which can fail but never sleeps).
- It shouldn't perform any I/O, unless the I/O is guaranteed not to block.

For information on handling hardware interrupts, see Chapter 7, "NeXTbus Device Drivers."

Building In Debugging Code

This section discusses functions that can help you debug in two ways: by displaying information while your server runs and by checking assumptions in your server. The functions and macros discussed in this section are explained further in Chapter 10.

Displaying Debugging Information

To display debugging information while your server runs, you have two choices: **kern_serv_log()** and the kernel **printf()** function. You should use **kern_serv_log()** instead of **printf()** whenever possible, since **printf()** slows the system and affects the timing of your server. You should use **printf()** only for unusual events that you need to see as soon as they occur and for events that are likely to result in a system panic.

Using kern_serv_log()

kern_serv_log() logs a message that a user process can later pick up and print out. The main advantages of **kern_serv_log()** are its quickness and reliability, even when called from an interrupt handler. However, if the system panics before the user process can pick up a log message, that message will be lost.

Your server supplies to **kern_serv_log()** the string to be logged and the priority at which it should be logged. Higher numbers correspond to higher priorities, but the exact interpretation of priority numbers is up to you.

Messages logged using **kern_serv_log()** can be obtained using either the kernel-server log command, **kl_log**, or any other user-level program that calls the functions **kern_loader_log_level()** and **kern_loader_get_log()**. By default, logging is off; you must set the log level before you can obtain any log messages. The **kl_log** utility is described in Appendix A, "Utilities for Loadable Kernel Servers"; **kern_loader_log_level()** and **kern_loader_get_log()** are described in Chapter 3, "Using Loadable Kernel Servers."

Using printf()

The kernel **printf**() function has the advantage that you can easily view its output. All you have to do is keep the console window open. However, **printf**() can greatly slow the system because nothing except hardware interrupt handling can happen during a call to **printf**(). Although **printf**() doesn't sleep, it's unreliable when called from an interrupt handler because messages can be garbled.

The kernel **printf**() function is best used when you have a short message that you want to see as soon as it happens. You can see **printf**() messages not only in the console window, but also in **/usr/adm/messages** and from a **msg** command in the NMI mini-monitor or Panic window. Your message is guaranteed to make it to the **msg** buffer (although it might be garbled) even if the kernel panics.

Checking Assumptions

To check assumptions in your server, you can use **ASSERT**() and **probe_rb**().

Using ASSERT()

ASSERT() evaluates the expression you pass to it. If the result of the expression is 0, **ASSERT**() prints a message describing the line and file that the assertion failed on, and then calls **panic**().

Note: **ASSERT**() doesn't do anything unless your server is compiled with the DEBUG C preprocessor macro defined. Other recommended compile flags are discussed in Chapter 9, "Building, Loading, and Debugging Loadable Kernel Servers."

Using probe_rb()

Use **probe_rb**() whenever you need to make sure that an address is valid. For example, to check whether a NeXTbus board is in a certain slot, you can call **probe_rb**(), passing the virtual address of one of the board registers.

Kernel-Server Loader Requirements

Your server must supply an instance variable to the kernel-server loader, **kern_loader**. You can also supply functions that **kern_loader** will call under certain circumstances, such as server initialization or shutdown.

You inform **kern_loader** of the instance variable name and of any functions to be called when you compile your server. This information goes into sections of your server's object file. See Appendix A for details on specifying information during compile time. The following sections first describe how to declare the instance variable in your code and then describe the types of functions you can write for **kern_loader** to call.

Instance Variable

Your server's instance variable is an ordinary C variable of type **kern_server_t** (defined in the header file **kernserv/kern_server_types.h**) that **kern_loader** uses to keep track of your server.

You can make the instance variable contain other information as well. Do this by defining a structure that begins with a field of type **kern_server_t**, followed by fields of your choice. For example, you might declare your instance variable in a header file as follows:

```
#import <kernserv/kern_server_types.h>

typedef struct my_instance_var
{
    kern_server_t  kern_server;  /* generic instance info */
    struct         my_dev;       /* per-device info */
    {
        int  field1;
        int  field2;
    } dev[MAX_MINE]
} my_instance_var_t;

my_instance_var_t  instance;
```

Writing Functions for kern_loader to Call

Your server can supply the following kinds of functions to **kern_loader**:

Kind of Function	Called When
Initialization	Your server is loaded.
Shutdown	Your server is unloaded.
Port server	Your server receives a message on a certain port.
Port death	A port for which your server has send rights dies.

Some servers might not require all or any of these functions. However, a server that doesn't start until it receives a message must supply **kern_loader** with the names and port servers of all ports that the server might receive its first message on.

Initialization functions can't be debugged with GDB. (You can't set a breakpoint in your server until it's fully loaded, and initialization functions are executed before then.) One way to get around this debugging problem is to have a message-based interface for initialization until your server is debugged. You can write a simple port server that initializes the server whenever it receives a message. You also have to write a simple program that sends this message. After you've finished debugging the initialization sequence, you can move it into the initialization functions called by **kern_loader**.

Shutdown functions are often used to free kernel resources. When the server is unloaded, no other part of the kernel can contain a reference to any code or data contained within the loadable server. If the kernel tries to refer to any code or data in an unloaded server, the system panics.

Considerations for Message-Based Servers

Message-based loadable kernel servers are built using the Mach Interface Generator (MiG). Examples of using MiG are in Chapter 2 of this manual, as well as under **/NextLibrary/Documentation/NextDev/Examples**. Under the **Examples** directory, the **MiG** example contains only user-level code, but it's a good starting place if you're unfamiliar with MiG. The **Log** example features a loadable kernel server that can receive only one type of request. The **ServerVsHandler** example has two versions of a loadable kernel server that can receive two types of requests.

Server versus Handler Interfaces

A message-based loadable kernel server can have one of two interfaces: a server interface or a handler interface. MiG automatically produces a server interface, but if requested it can produce most of the code for a handler interface. You have to write just a little more code to complete a MiG-generated handler interface. Handler interfaces have a performance advantage because they can use much less wired kernel memory if the loadable kernel server never returns much data.

For example, for a loadable kernel server with a server interface, MiG allocates reply messages that are MAX_SIZE_BYTES (currently 8192) bytes long. If the server returns only a little data—for example, if it has ten interface functions, each returning only an integer indicating the call's success—then a lot of kernel memory is being wasted. If the same server has a handler interface, MiG allocates only 32 bytes for each reply message (24 bytes for the header, and 8 bytes for the message body).

The difference in memory usage happens because MiG always generates large messages to return server data, but for handlers, it allocates just enough space to hold the largest message that's returned.

Note: From the caller's point of view and from the point of view of the loadable kernel server function that the MiG-generated interface calls, handler and server interfaces look identical. The difference between them is visible only in the options you specify to MiG, the files that MiG generates, and the extra bit of code you must write, as described in the following procedure.

To convert a server to a handler, while keeping the exact same interface to user processes, follow these steps (substituting the name of your server for *name* and "mydriver"):

1. Copy your loadable kernel server's ".defs" file to another directory and enter the following command at a UNIX shell prompt: **mig** *name*.**defs -handler** *name*Handler.c **-sheader** *name*Handler.h **-user** *name*User.c **-header** *name*.h. For example:

```
mymachine> mig mydriver.defs -handler mydriverHandler.c -sheader
mydriverHandler.h -user mydriverUser.c -header mydriver.h
```

2. This creates a group of files. Look in the *name***Handler.h** file for the *name*_**t** structure. Here's an example of what it looks like:

```
typedef struct mydriver {
    void    *arg;       /* argument to pass to function */
    int      timeout;   /* timeout for RPC return msg_send */

    /* Routine mydriver_do_log */
    kern_return_t (*mydriver_do_log) (
    void *server);
} mydriver_t;
```

3. Import *name***Handler.h** to your server and add a global variable of type *name*_t that maps user functions to the corresponding function in your server. The kernel and the kernel loader will use this structure to call your server's functions. For example, for a ".defs" file that has the line "routine mydriver_do_log(server: port_t);", you might have the following:

```
#import "mydriverHandler.h"
kern_return_t mydriver_do_log(port_t server);

mydriver_t mydriver_funcs = {
    0,
    100,            /* in milliseconds */
    (kern_return_t (*)(void *))mydriver_do_log
};
```

4. In the load commands script, change all instances of SMAP to HMAP, and change each function argument to the name of the mapping structure. For example:

```
SMAP          mydriver0 mydriver_server 0
```

becomes:

```
HMAP          mydriver0 mydriver_handler mydriver_funcs
```

5. Change your server's makefile so that it performs the MiG command described above and has the correct dependencies. For example:

```
MIGOUTPUT=mydriverUser.c mydriverServer.c mydriver.h
SERVER_OBJ= mydriver_main.o mydriverServer.o
.
.
.
mydriver_reloc: ${SERVER_OBJ} LoadCommands UnloadCommands
    kl_ld -n mydriver -l LoadCommands -u UnloadCommands -i instance
\
    -d mydriver_loadable -o $@ ${SERVER_OBJ}

${MIGOUTPUT}: mydriver.defs
    mig mydriver.defs
```

might become:

```
MIGOUTPUT=mydriverUser.c mydriverHandler.c mydriverHandler.h \
          mydriver.h
SERVER_OBJ= mydriver_main.o mydriverHandler.o
.
.
.
mydriver_reloc: ${SERVER_OBJ} LoadCommands UnloadCommands
    kl_ld -n mydriver -l LoadCommands -u UnloadCommands -i instance \
    -d mydriver_loadable -o $@ ${SERVER_OBJ}

${MIGOUTPUT}: mydriver.defs
    mig mydriver.defs -handler mydriverHandler.c \
    -sheader mydriverHandler.h -user mydriverUser.c \
    -header mydriver.h

${SERVER_OBJ}:    mydriverHandler.h
```

For another example of converting a server to a handler, see the files under **/NextLibrary/Documentation/NextDev/Examples/ServerVsHandler**.

Sending and Receiving Out-of-Line Data

A message-based server that receives out-of-line data doesn't have direct access to the data. The data is inaccessible because it appears in the address map of the server's task, but loadable kernel servers use the kernel's address map instead of their own task's map. To read out-of-line data, your server must use **vm_write()** to map all or part of the data into the kernel map. Similarly, to send out-of-line data, your server must first call **vm_read()**

to map the data into the address map of the server's task. Both **vm_write()** and **vm_read()** work on entire pages, so the data sent out-of-line must be page-aligned. However, as long as none of the data is wired down, the data isn't copied until it's written to.

Note: Don't use **copyin()** and **copyout()** in message-based servers. They work only in UNIX-style servers.

The following example shows how to send and receive out-of-line data in a loadable kernel server. See **/NextDeveloper/Examples/UNIX/LKSOutOfLineData** for complete examples of both sending and receiving out-of-line data.

```
/* Receive out-of-line data. */
kern_return_t mydriver_import(port_t server, user_data_t user_data,
unsigned int length)
{
    kern_return_t r;
    user_data_t   tmp, local_data;
    port_t        kernel_task;

    kernel_task = kern_serv_kernel_task_port();
    /* Allocate non-wired (for efficiency) pages for data. */
    r = vm_allocate((vm_task_t)kernel_task, &(vm_address_t)local_data,
        8192, TRUE);
    if (r != KERN_SUCCESS) {
        printf("Call to vm_allocate failed: %d\n", r);
        return KERN_FAILURE;
    }
    tmp = local_data;

    printf("\nmydriver_import:  Calling vm_write\n");
    r = vm_write((vm_task_t)kernel_task, (vm_address_t)local_data,
        (pointer_t)user_data, 8192);
    if (r != KERN_SUCCESS)
        printf("Call to vmwrite failed\n");

    else {
        printf("Trying to read out-of-line data.\n");
        printf("First data element is: %c\n", *tmp++);
        printf("Second data element is: %c\n", *tmp++);
        printf("Third data element is: %c\n", *tmp++);
        printf("Fourth data element is: %c\n", *tmp++);
        printf("Fifth data element is: %c\n", *tmp++);

        (void)strncpy(tmp, local_data, length);
        printf("Contents of out-of-line data: %s\n", tmp);
    }
```

```
    return KERN_SUCCESS;
}

/* Send out-of-line data. */
kern_return_t mydriver_export(port_t server, user_data_t *user_data,
unsigned int *length)
{
#define DATA "ACK"
    kern_return_t r;
    user_data_t local_data;
    port_t kernel_task;
    unsigned int tmp;

    kernel_task = kern_serv_kernel_task_port();
    /* Allocate non-wired pages for data. */
    r = vm_allocate((vm_task_t)kernel_task, &(vm_address_t)local_data,
        8192, TRUE);
    if (r != KERN_SUCCESS) {
        printf("Call to vm_allocate failed: %d\n", r);
        return KERN_FAILURE;
    }

    local_data = strcpy (local_data, DATA);
    *length = strlen(local_data);
    printf("\nmydriver_export: exporting this data: %s\n",
        local_data);

    /* Need to allocate space in our task's map. */
    r = vm_allocate((vm_task_t)task_self(),
        (vm_address_t *)user_data, 8192, TRUE);
    if (r != KERN_SUCCESS) {
        printf("Call to vm_allocate failed: %d\n", r);
        return KERN_FAILURE;
    }

    /* Make data accessible to our task's map. */
    r = vm_read((vm_task_t)kernel_task, (vm_address_t)local_data,
            8192, (pointer_t *)user_data, &tmp);
    if (r != KERN_SUCCESS) {
        printf("Call to vm_read failed: %d\n", r);
        return KERN_FAILURE;
    }

    return KERN_SUCCESS;
}
```

Considerations for UNIX-Based Servers

For a UNIX-based server, you must provide the proper entry points and insert the server into the appropriate device switch tables. This section lists the entry points you need, but doesn't cover most entry points in detail. If an entry point is not sufficiently covered here, see Egan and Teixeira's *Writing a UNIX Device Driver.*

Note: You should call NeXT Technical Support to receive the device major number to use. Getting the major number from NeXT will help ensure that your server works with other NeXT-supplied and third-party servers.

UNIX Device Entry Points

This section shows all the entry points that a character or block driver can provide.

Character-Device Entry Points

Entry points for character devices are defined in the header file **sys/conf.h** as shown below.

```
struct cdevsw
{
    int   (*d_open)();
    int   (*d_close)();
    int   (*d_read)();
    int   (*d_write)();
    int   (*d_ioctl)();
    int   (*d_stop)();
    int   (*d_reset)();
    int   (*d_select)();
    int   (*d_mmap)();
    int   (*d_getc)();
    int   (*d_putc)();
};
extern struct  cdevsw cdevsw[];
```

Field	Description
d_open	A pointer to the server function that handles an **open()** system call.
d_close	A pointer to the server function that handles a **close()** system call.
d_read	A pointer to the server function that handles a **read()** system call.
d_write	A pointer to the server function that handles a **write()** system call.
d_ioctl	A pointer to the server function that handles an **ioctl()** system call.
d_stop	Not supported for user-written servers.
d_reset	Not used.
d_select	A pointer to the server function that handles a **select()** system call. If your device is ready for reading or writing, this function should return true. If your device is *always* ready for reading and writing, you can specify **seltrue()** in the **cdevsw** table, which will make the kernel return true without calling your server.
d_mmap	A pointer to the server function that handles memory mapping of device space to user space. This function, which is typically found in frame buffers, must return the page number of the passed offset.
d_getc	Not supported for user-written servers; used for console devices.
d_putc	Not supported for user-written servers; used for console devices.

If your driver implements the *d_select* entry point, it usually handles the case that no data is immediately available. In this case, your driver must remember that someone is interested in the data, as well as whether more than one party is interested. In typical UNIX systems, a pointer to the **proc** structure of the interested party is saved. In NeXT systems, however, the driver must save a pointer to a thread. The **selthreadcache()** function stores this thread pointer and ensures that the thread won't go away before **selthreadclear()** is called. The **selthreadclear()** function should be called when the awaited device activity has taken place or when the device is closed. The following example illustrates how to use **selthreadcache()** and **selthreadclear()**.

```
typedef struct {
    ...
    void    *selread;
    void    *selwrite;
    void    *selexcep;
    int      collision : 1;
    ...
} Mystruct;

xyzselect(dev, flag)
{
    Mystruct *myinfo = Myinfo[minor(dev)];

    switch (flag) {
    case FREAD:
        if ( data not available ) {
            if ( selthreadcache(&myinfo->selread) )
                // more than one party...
                myinfo->collision = 1;
        }
        ...
        break;
    ...
    }
    ...
}

xyzint(dev)
{
    ...
    if (  read data && myinfo->selread )
        xyzwakeup(myinfo, FREAD);
    ...
}

xyzwakeup(myinfo, flag)
Mystruct *myinfo;
{
    if (flag & FREAD) {
        int oldpri = splxyz();    // may not be necessary
```

```
        if (myinfo->selread) {
            selwakeup(myinfo->selread, myinfo->collision);
            selthreadclear(&myinfo->selread);
            myinfo->collision = 0;
        }
        splx(oldpri);
    }
    ...
}

xyzclose()
{
    ...
    if (myinfo->selread)
        selthreadclear(&myinfo->selread);
    ...
}
```

Block-Device Entry Points

Entry points for block devices are defined in the header file **sys/conf.h** as shown below.

```
struct bdevsw
{
    int  (*d_open)();
    int  (*d_close)();
    int  (*d_strategy)();
    int  (*d_dump)();
    int  (*d_psize)();
    int  d_flags;
};
extern struct  bdevsw bdevsw[];
```

Field	Description
d_open	A pointer to the server function that handles an **open()** system call.
d_close	A pointer to the server function that handles a **close()** system call.
d_strategy	A pointer to the server function that eventually handles **read()** and **write()** system calls.
d_dump	A pointer to the server function that dumps physical memory to the swap device when the system is going down. Used only for devices that can be used for swapping.
d_psize	A pointer to the server function that returns the size of the swap partition for swap devices. Used only for devices that can be used for swapping.
d_flags	Contains flags that give more information about the device to the kernel. The only defined flag is B_TAPE, which tells the kernel that it can't reorder I/O to this server.

Inserting UNIX Servers into Device Switch Tables

If your server is entered through UNIX system calls, you must insert it into the appropriate device switch tables during your server's initialization. While debugging, you should do this through a message-based interface. Later, you can transfer this to an initialization function called by **kern_loader**.

The following example illustrates a server inserting itself into switch tables. Since the example is taken from a block driver, the server inserts itself into both the **bdevsw** and **cdevsw** tables. Character drivers have to insert themselves only into the **cdevsw** table.

Note: In the following example, MY_BLOCK_MAJOR and MY_RAW_MAJOR are device major numbers, which you should obtain from NeXT Technical Support.

```
/*
 * Example of a driver inserting itself into the block and character
 * device switch tables.
 */

#import <sys/conf.h>

extern int nulldev();
extern int nodev();
extern int seltrue();
#define nullstr 0

struct bdevsw my_bdevsw =  {
                int (*myopen)(),
                int (*myclose)(),
                int (*mystrategy)(),
                nodev,
                nodev,
                0 };
struct cdevsw my_cdevsw =  {
                int (*myopen)(),
                int (*myclose)(),
                int (*myread)(),
                int (*mywrite)(),
                int (*myioctl)(),
                nodev,
                nulldev,
                seltrue,
                nodev,
                nodev,
                nodev };

struct bdevsw my_saved_bdevsw;
struct cdevsw my_saved_cdevsw;

/* Save whatever entries were in the tables for our major numbers. */
my_saved_bdevsw = bdevsw[MY_BLOCK_MAJOR];
my_saved_cdevsw = cdevsw[MY_RAW_MAJOR];

/* Put my entries in the switch tables. */
bdevsw[MY_BLOCK_MAJOR]= my_bdevsw;
cdevsw[MY_RAW_MAJOR]  = my_cdevsw;
```

7 *NeXTbus Device Drivers*

NeXTbus is the interface bus supported by all NeXTcube and original NeXT computers. NeXTbus boards, such as the CPU and NeXTdimension™ boards, plug into slots in the back of these computers and communicate over the NeXTbus. As Figure 7-1 shows, NeXTbus computers have four slots, numbered 0, 2, 4, and 6. The CPU board is always in slot 0. The remaining slots are available for other NeXTbus boards.

Note: NeXTstation™ computers don't have a NeXTbus or any expansion slots.

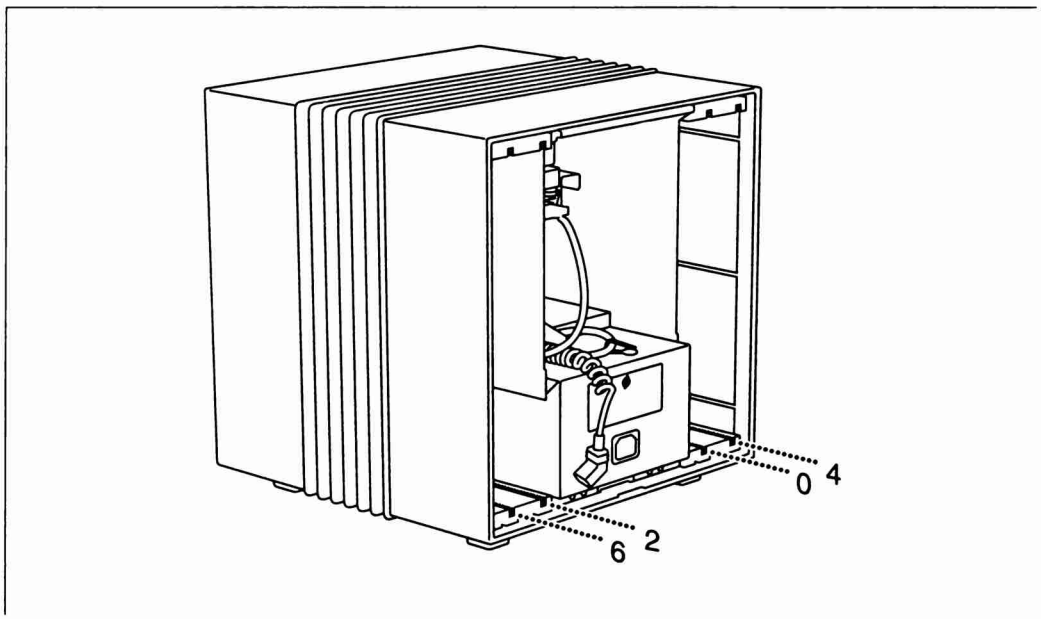

Figure 7-1. NeXTbus Slot Order

To be able to communicate with other NeXTbus boards, your computer's CPU board must have a NeXTbus Interface Chip (NBIC™). All NeXTcubes and 68040 Upgrade Boards have NBICs; however, original NeXT computers that still have the 68030 processor might not. See the *NeXTbus Development Kit Installation Guide* (part of the NeXTbus Development Kit) for instructions if you need to install an NBIC onto a 68030 board.

NeXTbus Address Space

Each NeXTbus board has access to two sections of the NeXTbus physical address space: the *board address space* and the *slot address space* that correspond to its slot. The board address space of each slot is 256 megabytes, from 0x*s*0000000 to 0x*s*fffffff, where *s* is the slot number; the slot address space of each slot is 16 megabytes at addresses 0xf*s*000000 to 0xf*s*ffffff. Every register that your driver must read or write appears in either the board or the slot address space; the specification for your hardware should tell you exactly which physical address you must use.

Warning: The addresses 0x*s*200c000 to 0x*s*200cfff, where *s* is 2, 4, or 6, are intercepted by logic on the CPU board; some CPU boards also intercept addresses 0x*s*1000000 to 0x*s*1ffffff, where *s* is 2, 4, or 6. Writing to these intercepted addresses can cause errors. If your driver needs access to information at these addresses, your board must be configured to accept addresses for the next higher slot, as described in the gray box later in this chapter.

Figure 7-2 shows the NeXTbus physical address space. As the figure shows, the NeXTbus architecture allows for up to 15 slots, each with up to 272 megabytes of physical address space.

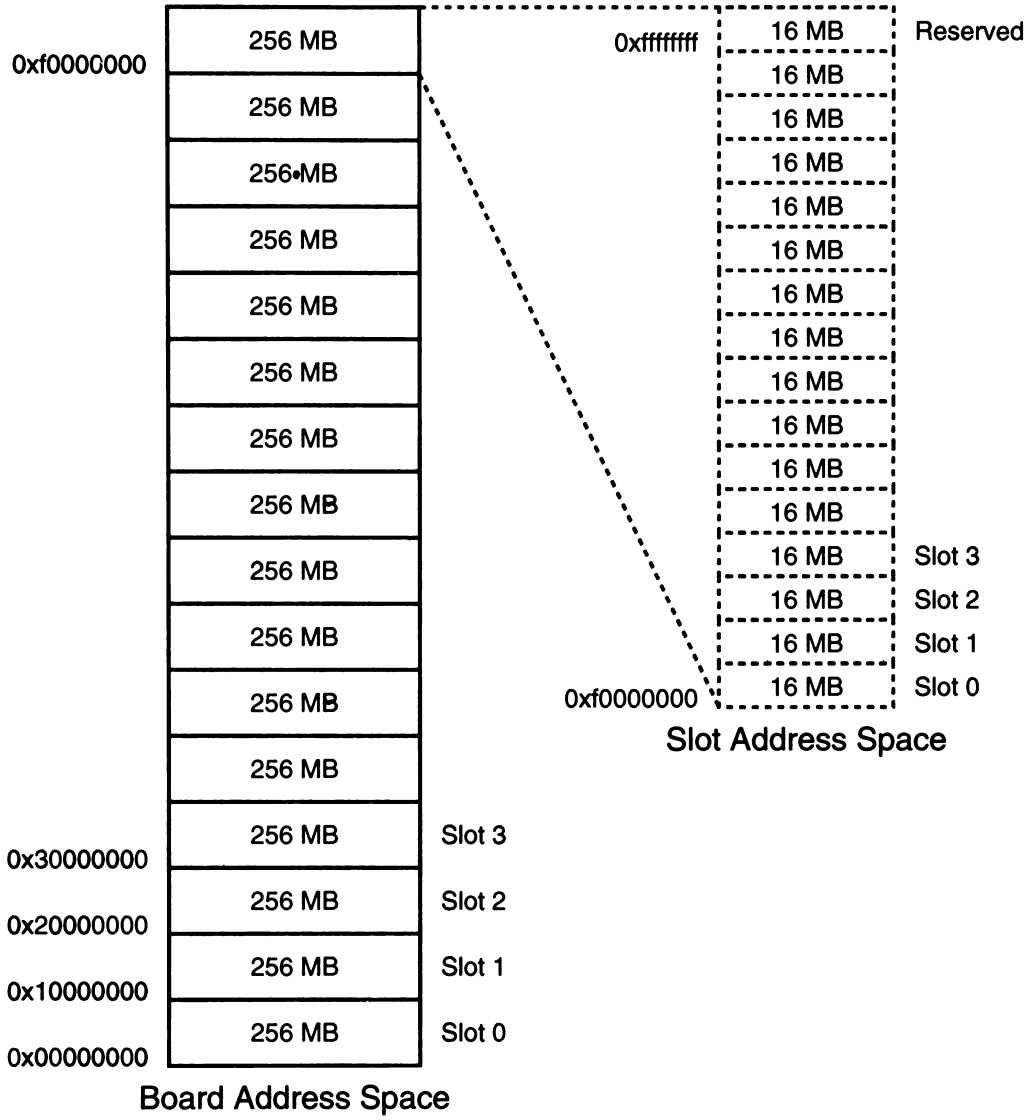

Figure 7-2. NeXTbus Address Space

To read or write a hardware address, your driver must first map the physical address into a virtual one. "Mapping from Physical to Virtual Memory Addresses," later in this chapter, describes how to perform this mapping.

Writing Code to Control the Hardware

This section discusses some of the considerations for writing code that controls NeXTbus hardware. Before you write any code, you should test the hardware with BusProbe™, an application in the NeXTbus Development Kit.

Using BusProbe

BusProbe is a NeXTSTEP application that lets you read and write any address on a NeXTbus board. BusProbe is useful in all phases of driver writing. When you first start, it can help you verify the address of each register. As you write your server, you can test sequences of reads and writes with BusProbe before you put the code into your server. Later, BusProbe can help you when you're testing or debugging your server.

You can use the repeat and autoincrement features to clear areas of memory on the board or for other repetitive tasks. BusProbe lets you choose the board or slot address space that you want to use. For this reason, you should specify an *offset* into this address space, not the whole address.

Before you're ready to write any code, try reading from and writing to every register on your NeXTbus board. This will help you verify the address and format of each register.

While you're writing your driver, you can use BusProbe to manually enter the actions that you think your driver will have to take. Once you've made sure the actions work, you can put the code to do them into your driver.

Mapping from Physical to Virtual Memory Addresses

Use the routine **map_addr()** to turn a physical address into a virtual memory address that your driver can use. For example, to set a flag on a 32-bit register at address 0x100 in the slot address space of a NeXTbus board (physical address 0x*fs*000100), you would use the following code:

```
#define REG_ADDR 0xf0000100
volatile unsigned int *reg;

reg = (unsigned int *)map_addr(REG_ADDR | (slot_number << 24), 4);
*reg |= FLAG;
```

For a description of the physical address space, see the previous section, "NeXTbus Address Space."

Transferring Data

NeXTbus boards can transfer data in up to three sizes: bytes (8 bits), halfwords (16 bits), and words (32 bits). You usually store bytes in **unsigned char** structures, halfwords in **unsigned short** structures, and words in **unsigned int** structures.

Warning: Systems based on the 68040 processor don't allow you to read or write using a larger data structure than the device width. For example, you can't write a 32-bit quantity into an 8-bit register; you must use an 8-bit quantity such as an **unsigned char**.

Burst transfers (transfers of 16 bytes with one instruction) are turned off by default for addresses that are mapped using **map_addr()**. (This is a side effect of **map_addr()** making the memory noncacheable, which is preferable when dealing with devices.) To perform a burst read or write on an address mapped with **map_addr()**, you must write assembly code that calls the 68040's MOVE16 instruction. Because MOVE16 reads or writes the bytes in 32-bit chunks, your hardware must be 32 bits wide. Note that the 68030 doesn't have a MOVE16 instruction and doesn't support burst writes. You can also enable burst transfers while using BusProbe by turning "Cache Inhibit" off.

Note: You can't currently use DMA (direct memory access) in a NeXTbus driver. Instead, you must use programmed (direct) I/O.

Workaround for Intercepted NeXTbus Addresses

If your driver needs to read or write the addresses that are intercepted by the CPU board (0x*s*200c000 to 0x*s*200cfff and 0x*s*1000000 to 0x*s*1ffffff, where *s* is 2, 4, or 6), the NBIC on your board must be configured to accept the addresses for the next higher slot (3, 5, or 7).

You can't directly configure the NBIC from your driver; it must be done locally on your board. Specifically, the board must set the IGNSID0 bit (bit 28) of its own NBIC Control register to 1. Then, in your driver, you can address both the slot and board address spaces of the board with the next higher slot number.

NeXTbus Byte Ordering

Depending on your hardware, bytes might be swapped within a word. For example, if you read a word on your NeXTbus board that contains "0x12345678", your driver might see "0x78563412".

You can determine whether byte order is an issue for your driver by consulting the designers of and documentation for your board, and by testing reads and writes using BusProbe. BusProbe does not change byte ordering, so if the reads and writes work with BusProbe, they should work the same way in your driver.

The code below is an example of reversing byte order. In this example, the programmer wants to check whether bit 7 in a word is 1 by performing a bitwise OR with the word and 0x00000080. However, on a board that swaps bytes, the programmer has to switch the first and fourth bytes (0x80 and 0x00), producing 0x80000000.

```
volatile unsigned int *my_register;
int reg_size = 4;   /* # of bytes in the space to be mapped */
my_register = (unsigned int *)map_addr((MY_ADDR | (slot_num << 24)),
    reg_size);

if (probe_rb(my_register))
    /* Make sure the Valid field is set */
    if ( !(*my_register & 0x80000000)) /* byte-reversed 00000080 */
        return FALSE;
```

Writing an Interrupt Handler

If your driver must directly detect interrupts, you must provide an interrupt handler. Your interrupt handler can be called whenever *any* NeXTbus board interrupts, not just when a board controlled by your driver interrupts.

For this reason, your interrupt handler must check whether your driver's hardware generated the interrupt. If it did, your driver must handle the interrupt, stop the hardware from generating this interrupt, and return true. If your driver's hardware did not generate the interrupt, your interrupt handler must do nothing and return false.

To stop NeXTbus hardware from generating an interrupt, write a 1 to bit 7 of the Interrupt Mask byte. This byte is discussed later in this chapter.

Warning: Your interrupt handler (and any function it might call) must not sleep. See Chapter 6 for more details on precautions that interrupt handlers must take.

Because interrupt handlers execute on behalf of the hardware, they have no knowledge of which user process they're working for. Thus, they can't use anything to do with a user process.

You install interrupt handlers using **install_polled_intr()**, and remove them using **uninstall_polled_intr()**.

The following example illustrates an interrupt handler.

```
#define SLOT_INTR_BIT 0x80
#define SLOTCOUNT 4
/* In the following macro, slotid is half the slot #. */
#define nbic_regs(slotid) (caddr_t)(0xf0ffffe8 | ((slotid)<<25)

mydriver_var_t my_var[SLOTCOUNT];
. . .
    sp = &my_var[slotid];
    sp->nbic_addr=(unsigned char *)map_addr(nbic_regs(slotid), 24);
. . .

int my_intr(void)
{
    int  slotid;
    volatile unsigned char *intr_reg, *mask_reg;
    /*
     * Figure out if we really handle this interrupt.
     * Interrupts at level 5 are polled, so we have to
     * check the interrupt byte associated with the
     * hardware slots we control.  slotid 1 corresponds
     * to hardware slot 2, slotid 2 to hardware slot 4,
     * and slotid 3 to hardware slot 6.
     */
    for (slotid = 1; slotid < SLOTCOUNT; slotid++)
    {
        if (my_var[slotid].is_ours) {
            intr_reg = my_var[slotid].nbic_addr;
            if (*intr_reg & SLOT_INTR_BIT)
                break;
        }
    }

    /* If we couldn't handle the interrupt, leave now. */
    if (slotid == SLOTCOUNT)
        return FALSE;        /* Poll code should try next handler */
```

```
    /*
     * At this point some device-dependent code is necessary to
     * reset the interrupt condition so that the device does
     * not continue to try to interrupt the CPU.  Here we
     * disable the interrupt by clearing the mask bit.
     */
    mask_reg = intr_reg + 4;
    *mask_reg = 0;  /* Mask it off. */

    /* Handle the interrupt. */


    /*  Schedule a routine to react to the interrupt.*/
    kern_serv_callout(&instance,
                      (void (*)(void *))my_func,
                      (void *)sp);

    return TRUE;              /* We fielded the interrupt, so no
                                 other driver should be polled. */
}

. . .
    install_polled_intr(I_BUS, my_intr);
```

NeXTbus Board Registers

This section discusses the standard NeXTbus registers that your driver might have to read or write. For more information on the NeXTbus, see the *NeXTbus Specification*. For information on the NeXTbus Interface Chip (NBIC), see the *NeXTbus Interface Chip Specification.*

CPU Board NBIC Registers

The only NBIC register on the CPU board that you might need to write to is the NBIC Control register, which has three defined bits. The only bit you might need to modify, however, is the Store Forward bit.

NBIC Control Register

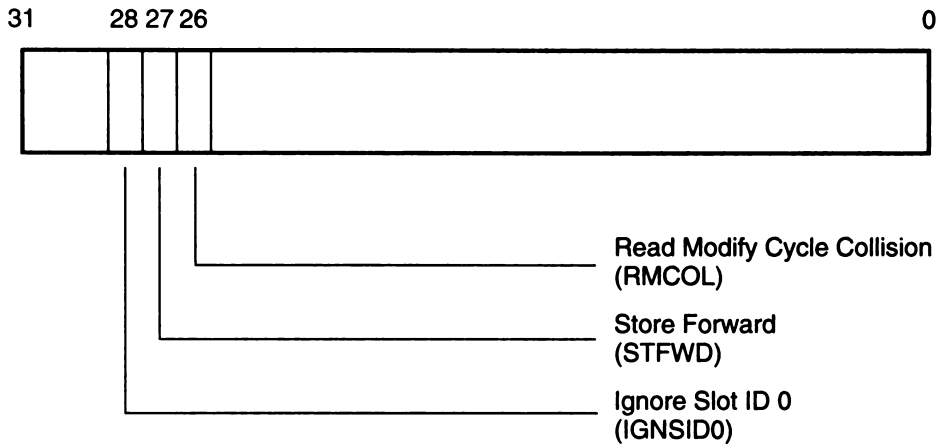

Figure 7-3. NBIC Control Register

The CPU board's NBIC Control register is at address 0x02020000.

Bit 28, Ignore Slot ID 0 (IGNSID0), controls how much NeXTbus address space a board uses. It's set to 1 during initialization so that the CPU board takes up two slots worth of addresses.

Bit 27 is the Store Forward (STFWD) control bit. At power up, it's enabled (set to 1). When enabled, Store Forward causes the CPU board's NBIC to immediately acknowledge writes, without waiting for the other NeXTbus board to write the data. This is called a *store and forward write* transaction. It speeds up transaction time, since the CPU doesn't have to wait for your board to write data. The disadvantage of store and forward write is that you won't receive any notification of write errors. Thus, unless your hardware is completely reliable, store and forward write can be dangerous.

Bit 26 is the Read Modify Cycle Collision (RMCOL) bit. It's not appropriate to access this register on the CPU board.

Other NeXTbus Board Registers

On every NeXTbus board, 6 bytes of identification and interrupt information are at addresses 0xf*s*fffffe8 to 0xf*s*fffffc, where *s* is the slot number of the board.

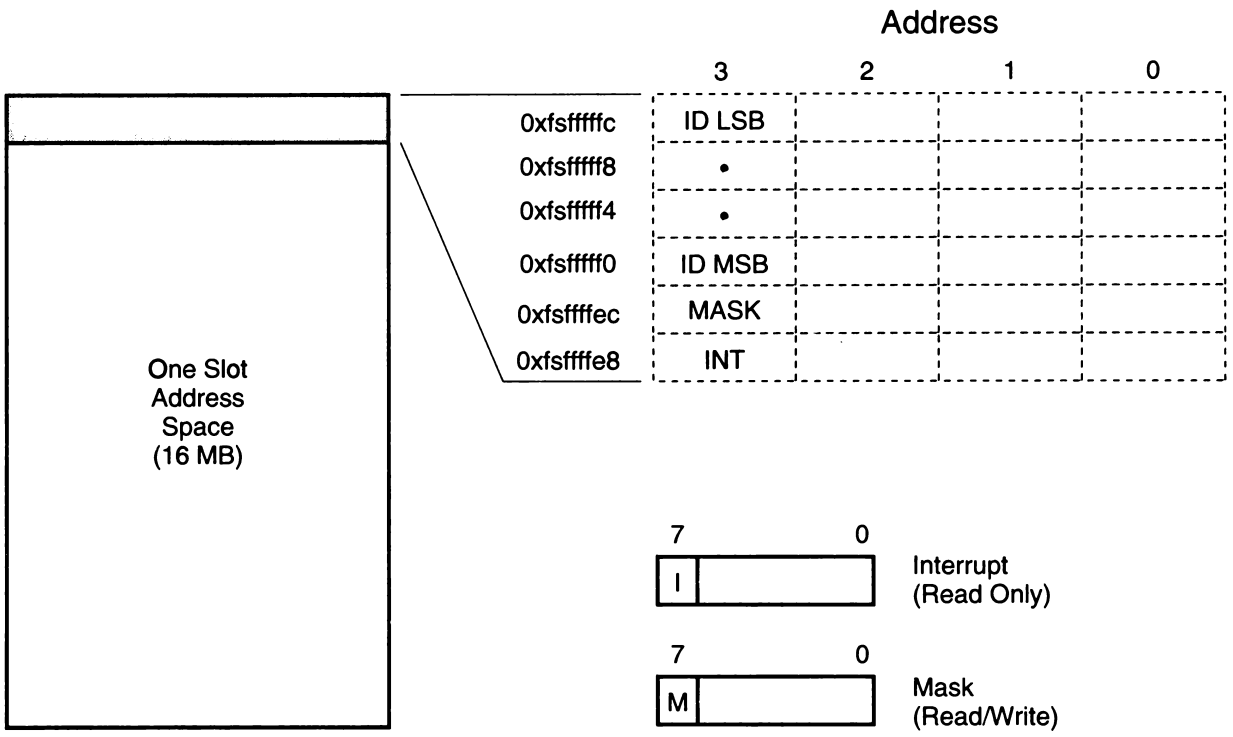

Figure 7-4. Sample NeXTbus Slot Address Space

If your NeXTbus board uses an NBIC, then addresses 0xf*s*ffffff0 through 0xf*s*fffffc correspond to the NBIC ID register, address 0xf*s*ffffec is the Interrupt register, and 0xf*s*fffffe8 is the Interrupt Mask register.

Identification Bytes (NBIC ID Register)

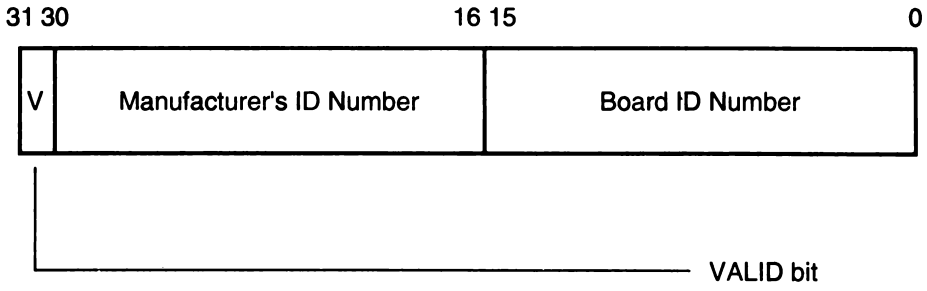

Figure 7-5. NBIC ID Register

Identification bytes give information that identifies the type of NeXTbus board. Your driver should read them during its initialization to see whether it should take control of this board.

The four Identification bytes are read-only bytes that contain a board ID number, a manufacturer's ID number, and a VALID bit. Unless the VALID bit is 1, none of the identification and interrupt information is valid. When the VALID bit is 1, the ID numbers are valid and your driver can use them to identify the board.

Since each byte is mapped to a separate 32-bit word, your driver has to read four separate words (or bytes, if your hardware supports byte-wide reads) to get all the identification information. Figure 7-6 shows the Identification byte locations in the NeXTbus slot address space.

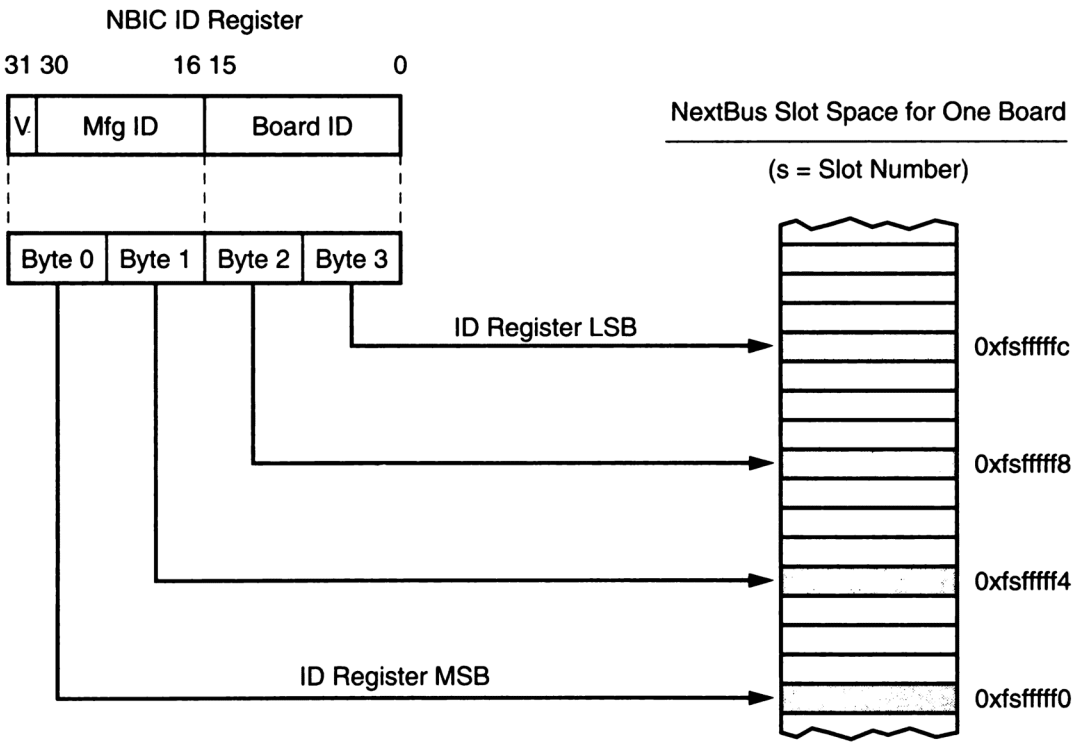

Figure 7-6. Identification Bytes in the NeXTbus Slot Space

If your driver can't perform byte reads, you have to find the correct byte in each word or halfword. If your driver doesn't have to reverse the byte order, you can simply address the last byte of the word as an **unsigned char**. If you do have to reverse the byte order, you must right-shift each word or halfword to move the byte you want to read from the most significant to the least significant byte.

The following example shows how to read the Identification bytes.

```
volatile unsigned int *id_start;
int id_size = 16;   /* # of bytes in the space to be mapped */

id_start=(unsigned int *)map_addr((0xf0fffff0 | (slotid << 25)),
    id_size);

if (probe_rb(id_start))
{
    if (ids_ok(id_start, MY_MFG_ID, MY_BRD_ID))
        my_var[slotid].present = TRUE;
}

#define HI_BYTE(n) ((n) >> 8)
#define LO_BYTE(n) ((n) & 0xff)

/*
 * Check the ID on the remote board to see if it matches ours.  The
 * ID fields are spread out over four words from 0xFsFFFFF0 through
 * 0xFsFFFFFC, with the valid data appearing in bits 0 through 7.
 */
boolean_t
ids_ok(volatile unsigned int *id_begin, int board_id, int mfg_id)
{
    volatile unsigned int *current_word = id_begin;

    /* Make sure the Valid field is set */
    if ( !(*current_word & 0x00000080))
        return FALSE;

    /* test high byte of mfg ID, ignoring the Valid field */
    if ( ((*current_word) & 0x7f) != HI_BYTE(mfg_id) )
        return FALSE;

    ++current_word;  /* Move to next word */
    /* test low byte of the mfg ID */
    if ( (*current_word) != LO_BYTE(mfg_id) )
        return FALSE;

    ++current_word;  /* Move to next word */
    /* test high byte of the board ID */
    if ( (*current_word) != HI_BYTE(board_id) )
        return FALSE;
```

```
        ++current_word;  /* Move to next word */
        /* test low byte of the board ID */
        if ( (*current_word) != LO_BYTE(board_id) )
            return FALSE;

        return TRUE;
    }
```

Interrupt Byte

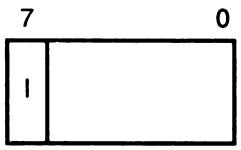

Figure 7-7. NBIC Interrupt Register

This byte contains a bit that shows whether the board wants to interrupt the CPU. Your driver's interrupt handler can read this bit to determine whether it needs to handle an interrupt.

The Interrupt byte is a read-only byte at address 0xfsfffe8, where *s* is the slot ID. This byte has only one significant bit, bit 7. Bit 7 is 1 when the board wants to interrupt, and 0 when it doesn't.

Note: The value of bit 7 doesn't depend on whether interrupts are enabled or disabled. Even if interrupts are disabled, and thus this board can't interrupt the CPU, bit 7 will be 1 if the board wants to interrupt.

Interrupt Mask Byte

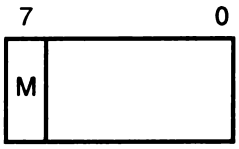

Figure 7-8. NBIC Interrupt Mask Register

Use this byte to enable and disable interrupts.

The Interrupt Mask byte is a read/write byte at address 0xf*s*fffffec, where *s* is the slot ID. As in the Interrupt byte, bit 7 is the only meaningful bit. When bit 7 is 1 (the default), the board can interrupt. If your driver writes a 0 to bit 7, the board stops interrupting and can't interrupt again until the driver writes a 1 to it.

After your driver handles an interrupt, you might want to disable interrupts for a while so that the hardware won't keep interrupting the CPU.

8 *Network Modules*

Loadable kernel servers that serve as network modules use special functions and interfaces in addition to the ones available to all loadable kernel servers. This chapter discusses how to write network modules. The special functions that network modules can use are described in detail in Chapter 10, "Kernel Support Functions," under the section "Network Functions."

The NeXT Mach kernel supports the following types of network modules:

- *Network device drivers*. A network device driver sends and receives packets to and from some network media.
- *Protocol handlers*. On input, a protocol handler receives packets from network device drivers and forwards the data to the interested programs. On output, the protocol handler takes data from programs, puts the data into packets, and sends these packets to the appropriate network device driver.
- *Packet sniffers*. A packet sniffer examines input packets for diagnostic purposes.

If you're familiar with UNIX 4.3BSD networking primitives, you'll find many similarities to what's described in this chapter. The biggest difference is that a common programming interface like the socket mechanism isn't defined. While sockets work well for TCP/IP, they don't generalize well to other protocols.

If you're writing a protocol handler and want to open it up to programmers, you must define your own interface for communication between user programs and your protocol handler.

This chapter first gives an overview of NeXT networking support, and then discusses the objects you'll use in your network module. The next section has details on the routines that you should implement. The chapter ends with notes on implementing specific interfaces.

Overview

Here's a simplified view of what happens when a network packet is received by a NeXT computer:

1. The packet is received by the appropriate network device driver, which puts the packet into a data structure called a *netbuf* (netbufs are discussed in the next section).

2. The driver calls the dispatcher (by calling **if_handle_input**()).

3. The dispatcher polls all registered packet sniffers and protocol handlers until it finds a protocol handler that accepts the packet.

4. If the protocol handler is an IP (Internet Protocol) handler, it sends the packet up to the kernel by calling **inet_queue**().

When a packet is sent out onto the network, the following events happen:

1. The output function of the protocol handler is called. One of the arguments is a netbuf containing the packet to be sent. (This netbuf must have been previously allocated by the network device driver; how netbufs are allocated is described later in this chapter.)

2. The protocol handler calls the appropriate network device driver's output function, passing it the netbuf containing the packet.

3. The device driver puts the packet out onto the network.

Note that there's one extra step in the case of the input packet: A dispatcher is called. This happens because a network device driver doesn't know what its associated protocol handler is, but the protocol handler knows which driver to call. The dispatcher doesn't query the modules in any particular order, except that it queries all packet sniffers before querying any protocol handlers.

Network Objects

The NeXT kernel includes two abstractions especially for network modules:

- Network buffers, known as *netbufs*
- Network interfaces, known as *netifs*

Each of these abstractions is discussed in the following subsections. The associated C functions are described in detail in Chapter 10.

Network Buffers (Netbufs)

The NeXT kernel uses netbufs for dealing with network packet buffers. Netbufs are an interface to an abstract sequence of bytes that can be read and written. The sequence has an original starting point and ending point, but these can be changed. An input network packet typically has its starting point advanced as the various headers are pulled off. Similarly, an output packet has its starting point retreated as headers are inserted.

Operating beyond the range of the original starting and ending points isn't currently detected as an error. This means that an outgoing netbuf should be copied into a larger netbuf if the information being added to its top requires more bytes than are available between the current ending point and the original starting point.

Network Interfaces (Netifs)

Netifs are used to handle the installation and usage of network modules. Remember that a network module is one of three things: a network device driver, a protocol handler, or a packet sniffer.

Each network module initializes and installs its netif (thus registering itself) by calling **if_attach()**. A network device driver should immediately register itself by calling **if_attach()** at load time. Protocol handlers and packet sniffers, on the other hand, don't have to register themselves until their services are required. They determine whether to register themselves in a callback function that they supply as an argument to the function **if_registervirtual()**. This callback function is called once for each network device driver; it should call **if_attach()** if the module isn't already registered and it wants to receive input packets from the specified driver.

Functions Implemented in the Network Module

Besides a callback function, your network module needs to supply certain functions so that other modules can call it. When your network module calls **if_attach()**, you must specify the locations of five functions:

- Initialization function—Does any initialization that's required to change the module's state to "on."
- Input function—Receives packets from lower layers and either consumes them or passes them on to other modules.
- Output function—Sends packets from higher layers.
- Getbuf function—Provides netbufs for higher layers to use in impending sends.
- Control function—Provides any necessary operations the above functions don't.

Note: You should specify null to **if_attach()** for any unimplemented function.

These five functions, along with the callback function, are described in more detail in the following subsections.

Callback Function

A *callback function* is required in protocol handlers and packet sniffers, but isn't appropriate in network device drivers. It must have the following syntax:

void *callback_func*(void **private*, netif_t *realif*)

The purpose of the callback function is to determine whether its network module is interested in a particular device driver and, if necessary, to register its module (using **if_attach()**). The callback function is called once for each current and future network device driver, so it can keep information about more than one network device driver.

The callback function is specified in the network module's call to **if_registervirtual()**. The *private* argument is the data that was specified in the call to **if_registervirtual()**. The *realif* argument is a pointer to the network device driver for which this function is being called. The following code is an example of a typical callback function for a protocol handler.

```
static void myhandler_attach(void *private, netif_t rifp)
{
    netif_t ifp;
    const char *name;
    int unit;
    void *ifprivate;

    if (strcmp(if_type(rifp), IFTYPE_ETHERNET) != 0) {
        return;
    }

    ifprivate = (void *)kalloc(sizeof(myhandler_private_t));
    name = MYNAME;
    unit = MYUNIT;
    ifp = if_attach(NULL, myhandler_input, myhandler_output,
        myhandler_getbuf, myhandler_control, name, unit, IFTYPE_IP,
        MYMTU, IFF_BROADCAST, NETIFCLASS_VIRTUAL, ifprivate);

    (myhandler_private_t *)if_private(ifp)->rifp = rifp;

    if_control(rifp, IFCONTROL_GETADDR, MYHANDLER_ADDRP(ifp));

    if (verbose) {
        printf("IP protocol enabled for interface %s%d, type
            \"%s\"\n", name, unit, MYDRIVER_TYPE);
    }
    return;
}

void myhandler_config(void)
{
    if_registervirtual(myhandler_attach, NULL);
}
```

Initialization Function

An *initialization function* is not required but is often found in network device drivers. It must have the following syntax:

int *init_func*(netif_t *netif*)

The initialization function takes a pointer to its module's netif structure and performs any necessary initialization. For example, a network device driver should perform any steps necessary to have its hardware ready to run. You can determine what the integer return value (if any) should be. The following is an example of an initialization function.

```
int mydriver_init(netif_t netif)
{
    unsigned unit = if_unit(netif);
    register struct mydriver_data_t *is = &mydriver_data[unit];

    if (is->is_flags & HW_RUNNING)
        return;

    is->is_flags |= HW_RUNNING;
    /* Initialize software structures and the hardware. */
    /* ... */
    return;
}
```

Input Function

An *input function* is required in protocol handlers and packet sniffers, but not in network device drivers. It must have the following syntax:

int *input_func*(netif_t *netif*, netif_t *realnetif*, netbuf_t *packet*, void **extra*)

The input function takes a pointer to its module's netif (*netif*), a pointer to the calling network device driver (*realnetif*), the input packet, and optional extra data. This function should examine the input packet and decide if it wants the packet. If so, this function should return zero and take responsibility for freeing the packet. Otherwise, this function should return EAFNOSUPPORT to allow other modules to receive the packet. Packet sniffers should always return EAFNOSUPPORT.

For example, an IP handler getting packets from an Ethernet device would check if an Ethernet packet's protocol number is the value for IP. If so, the IP handler should handle the packet and return zero.

Since this function might be called at interrupt priority, it should only queue packets. Another thread should pull the packets off of the queue and process them.

The following code is a typical input function of a protocol handler.

```
static int venip_input(netif_t ifp, netif_t rifp, netbuf_t nb,
    void *extra)
{
    short etype;
    short offset;
    short size;
    trailer_data_t trailer_data;

    /* Do we want packets from this driver? */
    if ((myhandler_private_t *)if_private(ifp)->rifp != rifp) {
        return (EAFNOSUPPORT);
    }

    /*
     * Check fields in the packet to see whether they match
     * the protocol we understand.
     */
    nb_read(nb, MYTYPEOFFSET, sizeof(etype), &etype);
    etype = htons(etype);
    /*
     * Handle ethernet trailer protocol.
     */
    if (etype >= ETHERTYPE_TRAIL &&
        etype < ETHERTYPE_TRAIL + ETHERTYPE_NTRAILER) {
        offset = (etype - ETHERTYPE_TRAIL) * 512;
        if (offset == 0 || (ETHERHDRSIZE + offset +
                    sizeof(trailer_data) >=
                    nb_size(nb))) {
            return (EAFNOSUPPORT);
        }
        nb_read(nb, ETHERHDRSIZE + offset, sizeof(trailer_data),
            &trailer_data);
        etype = htons(trailer_data.etype);
        if (etype != ETHERTYPE_IP &&
            etype != ETHERTYPE_ARP) {
            return (EAFNOSUPPORT);
```

```
            }
            size = htons(trailer_data.length);
            if (ETHERHDRSIZE + offset + size > nb_size(nb)) {
                return (EAFNOSUPPORT);
            }
            /*
             * trailer_fix() is a private function that converts trailer
             * packet to regular ethernet packet.
             */
            trailer_fix(nb, offset, size - sizeof(trailer_data));
        }
        switch (etype) {
        case ETHERTYPE_IP:
            nb_shrink_top(nb, ETHERHDRSIZE);
            if_ipackets_set(ifp, if_ipackets(ifp) + 1);
            inet_queue(ifp, nb);
            break;
            /* Put other cases here as necessary. */
            default:
            /*
             * Do not free buf:  let others handle it
             */
            return (EAFNOSUPPORT);
        }
        return (0);
    }
```

Output Function

All network modules except packet sniffers must have an *output function*. The syntax of this function must be the following:

int *output_func*(netif_t *netif*, netbuf_t *packet*, void **address*)

The output function takes a pointer to the module's netif, a pointer to a packet, and an address. How this function works depends on whether it's part of a protocol handler or of a network device driver.

If this function is part of a protocol handler, it should assume the packet and address are strictly protocol-level entities, containing no device-dependent information. The function should add network device information to the packet and call the network device driver's output routine. The netbuf that holds the packet should have been returned by this module's getbuf function, as described later in this chapter.

If this function is part of a network device driver, it should assume the packet and address are device-level entities. The function should simply deliver the packet to the given device-level address. Its return value should be zero if no error occurred; otherwise, return an error number from the header file **sys/errno.h**.

The following example illustrates a typical output function.

```
static int venip_output(netif_t ifp, netbuf_t nb, void *addr)
{
    struct sockaddr *dst = (struct sockaddr *)addr;
    struct ether_header eh;
    struct in_addr idst;
    int off;
    int usetrailers;
    netif_t rifp = VENIP_RIF(ifp);
    int error;

    switch (dst->sa_family) {
    case AF_UNSPEC:
        bcopy(dst->sa_data, &eh, sizeof(eh));
        break;
    case AF_INET:
        idst = ((struct sockaddr_in *)dst)->sin_addr;
        /* ... */
        /*
         * Resolve the en address using arp.  Return 0 if the address
         * wasn't resolved.
         */
         /*
         * XXX:  trailers not supported for output
         */
        eh.ether_type = htons(ETHERTYPE_IP);
        break;
    default:
        nb_free(nb);
        return (EAFNOSUPPORT);
    }
    nb_grow_top(nb, ETHERHDRSIZE);
    nb_write(nb, ETYPEOFFSET, sizeof(eh.ether_type),
         (void *)&eh.ether_type);
    error = if_output(rifp, nb, (void *)&eh.ether_dhost);
    if (error == 0) {
        if_opackets_set(ifp, if_opackets(ifp) + 1);
    } else {
        if_oerrors_set(ifp, if_oerrors(ifp) + 1);
    }
    return (error);
}
```

Getbuf Function

A *getbuf function* is required in all modules except packet sniffers. It must have the following syntax:

netbuf_t *getbuf_func*(netif_t *netif*)

This function returns a netbuf to be used for an impending output call. Only network device drivers should allocate these netbufs. Protocol handlers should instead call the appropriate network device driver's getbuf function to do the allocation. After allocation from the network device driver and before returning the result, the protocol handler should leave enough room at the top of the netbuf for its own output function to later insert a header.

A getbuf function doesn't always have to return a buffer. For example, you might want to limit the number of buffers your module can allocate (say, 200 kilobytes worth) so that it won't use up too much wired-down kernel memory. When a getbuf function fails to return a buffer, it should return null.

In a protocol handler:

```
static netbuf_t venip_getbuf(netif_t ifp)
{
    netif_t  rifp = VENIP_RIF(ifp);
    netbuf_t nb;

    nb = if_getbuf(rifp);
    if (nb == NULL) {
        return(NULL);
    }
    nb_shrink_top(nb, ETHERHDRSIZE);
    return(nb);
}
```

In a driver:

```
static netbuf_t engetbuf(struct ifnet *ifp)
{
    if (numbufs == MAXALLOC)
        return(NULL);
    else {
        numbufs++;
        return(nb_alloc(HDR_SIZE + ETHERMTU));
    }
}
```

Control Function

The *control function* isn't required, but it's useful in all three kinds of network modules. It must have the following syntax:

int *control_func*(netif_t *netif*, const char **command*, void **data*)

The control function performs arbitrary operations; the character string *command* is used to select between these operations. There are five standard operations that you can choose to implement, although you can also define your own. The command strings corresponding to the standard operations are listed in the following table; constants for the strings (such as IFCONTROL_SET_FLAGS for "setflags") are declared in the header file **net/netif.h** (under the **bsd** directory of **/NextDeveloper/Headers**).

Command	Operation
"setflags"	Request to have interface flags turned on or off. The *data* argument for this command is of type **union ifr_ifru** (which is declared in the header file **net/if.h**).
"setaddr"	Set the address on the interface.
"getaddr"	Get the address of the interface.
"autoaddr"	Automatically set the address of the interface.
"unix-ioctl"	Perform a UNIX **ioctl()** command. This is only for compatibility; **ioctl()** isn't a recommended interface for network drivers. The argument is of type **if_ioctl_t ***, where the **if_ioctl_t** structure contains the UNIX ioctl request (for example, SIOCSIFADDR) in the **ioctl_command** field and the ioctl data in the **ioctl_data** field.

An example of a control function follows.

```
static int
venip_control(netif_t ifp, const char *command, void *data)
{
    netif_t rifp = VENIP_RIF(ifp);
    unsigned ioctl_command;
    void *ioctl_data;
    int s;
    struct sockaddr_in *sin = (struct sockaddr_in *)data;
```

```
    if (strcmp(command, IFCONTROL_AUTOADDR) == 0) {
        /*
         * Automatically set the address
         */
        if (sin->sin_family != AF_INET) {
            return (EAFNOSUPPORT);
        }
        /* ... */
    } else if (strcmp(command, IFCONTROL_SETADDR) == 0) {
        /*
         * Manually set address
         */
        if (sin->sin_family != AF_INET) {
            return (EAFNOSUPPORT);
        }
        if_flags_set(ifp, if_flags(ifp) | IFF_UP);
        if_init(rifp);
        VENIP_PRIVATE(ifp)->vp_ipaddr = sin->sin_addr;
        /* ... */
    } else {
        /*
         * Let lower layer handle
         */
        return (if_control(rifp, command, data));
    }
    return (0);
}
```

Notes for Specific Interfaces

This section contains notes about implementing Ethernet and TCP/IP interfaces.

Ethernet Interfaces

Network device drivers that implement the 10-megabit-per-second Ethernet protocol should register their type as IFTYPE_ETHERNET (defined in the header file **net/etherdefs.h**). One 10-megabit Ethernet network device driver comes standard with the NeXT operating system. The type of the address passed to the Ethernet driver's output function for output should be a 6-byte character array (which is cast to **void ***).

TCP/IP Interfaces

IP protocol handlers can hand over their input packets to the kernel for processing by calling **inet_queue()**.

IP protocol handlers should specify their type as "Internet Protocol" when they call **if_attach()**. The NeXT operating system comes with two TCP/IP modules—one for delivery over Ethernet, and one for delivery over loopback. The type of address used by IP protocol handlers should be **struct sockaddr_in**, which is defined in the header file **netinet/in.h**.

9 *Building, Loading, and Debugging Loadable Kernel Servers*

The process of writing a loadable kernel server typically includes many cycles of writing, building, loading, and debugging the code. This chapter explains how to build, load, and debug your server, as well as how to build and load it once it's ready for normal use. Much of this chapter is devoted to explaining how you can use the GNU source-level debugger, GDB, to debug a server.

Building a Loadable Kernel Server

To build a server, you need two tools: the GNU C compiler (**cc**) and the kernel-server linker (**kl_ld**). The following example is an excerpt from a typical Makefile for a loadable kernel server. It shows how to build either a debugging version or a final version of the server, using **cc** and **kl_ld**. (**kl_ld** is discussed in detail in Appendix A, "Utilities for Loadable Kernel Servers"; **cc** is discussed in the *NeXTSTEP Development Tools and Techniques* manual.)

```
# defaults for a final version of the server:
OFILE_DIR= ./obj
G_LOADABLE=
CFLAGS= -DKERNEL -DMACH_USER_API -DMACH -Wall -O2

# defaults for all versions of the server:
NAME=mydriver
CFILES= mydriver.c
OFILES= $(CFILES:.c=.o) $(MIGCFILES:.c=.o)
VPATH = $(OFILE_DIR)
```

```
debug:
    make $(NAME)_reloc_g  \
    "OFILE_DIR=./debugObj"  \
    "G_LOADABLE=-d $(NAME)_loadable" \
    "CFLAGS=-DKERNEL -DMACH_USER_API -DMACH -Wall -g -DDEBUG"

$(OFILE_DIR):
    mkdirs $@

$(NAME)_reloc $(NAME)_reloc_g: $(OFILE_DIR) $(OFILES)\
        loadCmds unloadCmds
    kl_ld -n $(NAME) -l loadCmds -u unloadCmds -i instance \
        $(G_LOADABLE) -o $@ $(OFILES)

.c.o:
    $(CC) $(CFLAGS) -c $*.c -o $(OFILE_DIR)/$*.o
```

The definitions of KERNEL, MACH_USER_API, and MACH (as shown above) are necessary for every loadable kernel server. They ensure that your server has access both to kernel functionality and to most of the Mach kernel functions that are listed in Chapter 4, "Mach Functions."

The **-Wall** option is not necessary, but it's a good way to check your code during compilation. It instructs the compiler to warn you of many possible errors, such as passing the wrong kind of argument to a function.

The **-O2** option causes the compilation to be optimized to produce a smaller object file that executes more quickly. Optimization generally isn't used until the server has been completely debugged because it can make debugging more difficult (by making variables disappear or the flow of control briefly move where you don't expect it).

Note: If you experience a bug that occurs only when the server is compiled with optimization, make sure that all variables that refer to hardware addresses and all variables used in more than one thread are declared as **volatile**.

The **-g** option to the compiler creates symbols in the server's object file, which makes it possible to see C source code while debugging with GDB. Released drivers usually aren't compiled with **-g** because it makes the object file much larger.

The definition of DEBUG is often used in debugging versions so that **ASSERT()** is defined.

Finally, the **-d** option to **kl_ld** specifies that **kern_loader** should create a loadable object file when it allocates the server. This file is necessary when debugging the server with GDB.

Loading a Server into the Kernel

Loadable kernel servers must be allocated against the kernel before they can be loaded into it. There are three ways to allocate a server:

- Add a line to the kernel-server loader configuration file, **/etc/kern_loader.conf**; the server will be allocated shortly after the system boots. The MIDI driver works this way. The kernel-server loader configuration file is discussed in Appendix A.

- Write a user-level program, such as an application or background task, that calls **kern_loader_add_server()**. For example, the InstallTablet application allocates the tablet driver when the user presses a button. The **kern_loader_add_server()** function is discussed in Chapter 3, "Using Loadable Kernel Servers."

- Use the command-line utility **kl_util** with the **-a** option. This command is especially useful while you're debugging your server. It can also be called from **/etc/rc.local**, a script called from **/etc/rc** during system initialization. For example, you could allocate your server depending on the result of a conditional statement in **/etc/rc.local**. The **kl_util** command is discussed in Appendix A.

Once a server is allocated, it can be loaded in these ways:

- As soon as it is allocated. To specify that this should happen, put a START command in the load commands script that's passed to **kl_ld**. For example, all UNIX-style servers must start up as soon as they're allocated, since they're accessed by table lookups instead of Mach messages. The load commands script is described in Appendix A.

- As soon as it receives a message on one of its ports. The ports that can receive the first message must be specified in the server's load commands script. Specifically, each port must be advertised using an ADVERTISE command and mapped to a handling function using an HMAP or SMAP command.

- By a user-level program that calls **kern_loader_load_server()**. The **kern_loader_load_server()** function is discussed in Chapter 3.

- By the **-l** option to **kl_util**.

When choosing how to load your driver, you should try to combine maximum user convenience with a minimum effect on system performance. For example, if you allocate and load your server just after the system boots (using **/etc/kern_loader.conf** and the START load command), you're increasing the amount of time that the system will take to become usable after booting. This extra time might be worthwhile if the user definitely needs the driver, but it's a high price to pay if the user doesn't need the driver. On the other

hand, if you don't load your server until it's called, then the first call to the server will take a couple of seconds longer than usual.

Another possibility might be that the user explicitly load the server, such as by pressing a button in an application. However, since this is probably an inconvenience to the user, you should have a good reason for requiring it.

Tools for Debugging Servers

The GNU source-level debugger, GDB, is the primary tool for debugging loadable kernel servers. However, you'll also use many other tools while debugging your server. This section describes these other tools. The section after discusses in detail how to use GDB to debug loadable kernel servers.

Generating Interrupts from the Keyboard

At times, your system might freeze due to bugs in your server or mistakes while running GDB. If this happens and you don't see a Panic window, first try to generate a non-maskable interrupt (NMI), as described below. If this doesn't work, then as a last resort you can reset the CPU.

To generate an NMI on a keyboard that has only one Command key, use Command-Left Alternate-~. (Hold down both the Command key and the leftmost Alternate key and press the ~ key. Do *not* press the Shift key.) To generate an NMI on a keyboard that has two Command keys, use Command-Command-~. (Hold down both Command keys and press the key at the upper left of the numeric keypad.)

After you generate an NMI, the NMI mini-monitor window appears. Besides the **msg** and **continue** commands discussed in the section "NMI Mini-Monitor Window," you can also enter a **reboot** or **halt** command to reboot or halt the computer, or a **monitor** command to enter the ROM monitor.

To reset the CPU, hold down the Command and Alternate keys at the lower left of the keyboard, and press the * key on the upper right of the numeric keypad. This causes the machine to reboot immediately. Rebooting will take longer than usual because the file system will be checked.

Warning: Resetting the CPU can damage the file system and should be used only as a last resort!

NMI Mini-Monitor Window

The NMI mini-monitor window is useful for looking at the output of kernel **printf()** calls, for rebooting the system, and, if the system fails to reboot, for entering the ROM monitor. To bring up the NMI mini-monitor window, generate a non-maskable interrupt (NMI) as described in the gray box titled "Generating Interrupts from the Keyboard."

To view the output of kernel **printf()** calls, use the **msg** command. You can limit the number of messages you see by putting *=n* after the command, where *n* is the number of messages you want to see. For example:

```
nmi> msg=3
```

To return to normal system operation, enter the **continue** command.

The **gdb** NMI mini-monitor command is sometimes necessary when using GDB to debug a server. This command is described later in this chapter, in the section "Debugging Servers with GDB."

For more information on the NMI mini-monitor window, see the *NeXTSTEP Network and System Administration* manual.

ROM Monitor Window

The ROM monitor window lets you specify command options for rebooting your system. For example, the **bsd -p** command boots the system and tells it not to automatically reboot if a system panic occurs. Boot commands and other generally useful ROM monitor commands are discussed in detail in the *Network and System Administration* manual.

Some less widely used ROM monitor commands, such as those that let you examine and change the contents of hardware addresses on the CPU board, are described in Appendix B, "The ROM Monitor and NMI Mini-Monitor."

Panic Window

By specifying the **-p** option at boot time, you tell the computer to bring up a Panic window instead of rebooting when the system panics. The Panic window is similar to the NMI mini-monitor window, but it comes up only as the result of a kernel panic. In this window you can use the **gdb** and **msg** commands, just as in the NMI mini-monitor window.

If the system brings up a Panic window, you can run GDB on the panicked system if it's connected to the network. To do so, start GDB as usual on the master system (including entering the **kattach** command, as discussed later in this chapter). You can then use the **backtrace** GDB command to see what function caused the panic.

After you're done using GDB, enter **halt** at the Panic window. Once the system has shut down, you can boot it. (If the system doesn't respond to entering **halt** and pressing Return, then try to shut down by using the **monitor** command to enter the ROM monitor.) Booting after a panic will take longer than usual, since the file system will be checked.

System Console

You can view the output of **printf()** statements without using the NMI mini-monitor by keeping the system console window open. To open a console window, choose the Console command from the Tools menu of Workspace Manager™.

Another way to use the console is to log in with the user name **console** (without a password). This will make the whole screen act like a UNIX terminal that receives all console messages. After you log in, you can enter commands at the shell prompt.

Logging in as **console** is useful when the only thing you want to run is your server, so you don't need the overhead of a windowing environment. This is sometimes true for the slave computer when you're debugging your server with GDB.

Other Debugging Tools

With BusProbe (an application in the NeXTbus Development Kit), you can read or write registers while you're testing or debugging a NeXTbus driver. BusProbe is described in Chapter 7, "NeXTbus Device Drivers."

The kernel-server log command, **kl_log**, is useful for getting log messages from your server. If you wish, you can instead write a program that calls **kern_loader_log_level()** and **kern_loader_get_log()**. Chapter 6, "Designing Loadable Kernel Servers," describes how loadable kernel servers create log messages.

The kernel-server utility, **kl_util**, not only lets you load and unload servers; it also can give you information about all servers or one server in particular. Entering **kl_util -s** gives you general information on all the servers that **kern_loader** knows about. To get more detailed information on a particular server, such as where in memory it's loaded, use **kl_util -s** *servername*.

The **vm_stat** command is useful for seeing how many pages are wired down. It can help you find out whether your server is growing too large. For information on **vm_stat**, see its UNIX manual page.

Debugging Servers with GDB

GDB, the GNU source-level debugger, is an important debugging tool. You can use it not only during the normal debugging cycle, but also later to examine kernel panics caused by your server. With GDB, you can debug every function in your server that's called after your server is loaded. However, you can't debug functions that are called when **kern_loader** is initializing your server.

To use GDB to debug your server, you need two computers that are connected to a network and are running the same major release of NeXTSTEP. GDB runs on one computer (the *master*), debugging the server that's running on the other computer (the *slave*). Once you have two computers, follow these steps to debug your server (as described in detail below):

1. Set up the computers.
2. Put the appropriate files where GDB can find them.
3. Start up and initialize GDB.
4. Debug with GDB.
5. Shut down GDB.

This section describes the special steps you must take to use GDB on a loadable kernel server. For general information on using GDB, see *NeXTSTEP Development Tools*. Most GDB commands work for loadable kernel servers. However, commands for listing and affecting threads don't work.

Setting Up the Computers

When setting up the master and slave computers, you must first choose which one will be the slave. Keep in mind that the slave computer will panic often and its operating system will often be frozen. Some of the consequences include:

- The slave computer should have a minimum of disk space, so that rebooting after a panic won't take too long (due to checking the disk).
- No one can depend on the slave computer working all the time. For example, you should not use the slave as an NFS® server.

- If you might need any files from the slave while it's panicked or frozen, be sure to copy them to another computer.

Of course, the slave computer should also support whatever hardware you need to test your driver.

Once you've decided which computer is the slave, set up both computers so that they're connected to the network. You'll need access to both keyboards, so put the computers close together.

Once you've set up the computers, boot them both. Use the **-p** option to the **boot** command on the slave computer so the Panic window will stay up.

To verify that the two machines can communicate with each other, use the **ping** command at a shell prompt. If the output doesn't include the phrase "0% packet loss," contact your system administrator. Below is an example of the output of **ping** when the network connection is working.

```
master> /usr/etc/ping slave
PING slave: 56 data bytes
64 bytes from 192.42.172.1: icmp_seq=0. time=13. ms
64 bytes from 192.42.172.1: icmp_seq=1. time=5. ms
<Control-C>
----slave PING Statistics----
2 packets transmitted, 2 packets received, 0% packet loss
round-trip (ms)  min/avg/max = 5/9/13
master>
```

Setting Up the Files

Now that you've set up the computers, you need to decide where to put the files that they need and how to keep them in up-to-date on both the master and slave computers.

Usually, files are kept up-to-date by using NFS to mount the directory containing the relocatable object file onto the slave computer. Another solution is to make the slave the NetBoot client of the master. Using NFS and NetBoot is covered in the *Network and System Administration* manual.

Files Needed by the Slave Computer

The slave computer needs only whatever files are required by **kern_loader**. Usually, this is just your server's relocatable object file. You must create the relocatable object file by compiling with the **-g** option so that it contains debugging information. Avoid using the **-O** option, since optimization can make variable values appear incorrect.

Files Needed by the Master Computer

The master computer needs access to the following:

- The directory that contains the source files for your server.
- Your server's loadable object file. This file is produced by **kern_loader** on the slave computer when your server is allocated, but only if you specify the location of this file when you link your server by using the **-d** option of the **kl_ld** command. (Information on using **kl_ld** is at the beginning of this chapter and in Appendix A.)
- A file that contains the same version of the kernel as the one that the slave is running. If the master and the slave are running the same version of the kernel, then you can use the **/mach** file on the master. You can check the version by searching for "mk-" in **/usr/adm/messages**.

Before you go on to the next step, write down the full name (using pathnames starting at the master computer's root directory) of the loadable object file and the server's source directory. You'll need to supply these names to GDB later.

For example, in the rest of this chapter, the loadable object file is **/me/mydriver/LKS/mydriver_loadable** and the server source directory is **/me/mydriver/LKS**.

Starting Up GDB

On the master computer:

1. Become **root** at a shell prompt.

```
master> su
Password:
```

2. Change to the directory containing the kernel file that's the same version as the one running on the slave.

```
master:1# cd /
```

3. Start GDB, specifying the name of the kernel file.

```
master:2# gdb mach
Reading symbol data from mach...
(no debugging symbols found)...done.
(gdb)
```

On the slave computer:

4. Load the server, if it isn't already running. (In the following example, a **kl_util -l** command isn't necessary after entering **kl_util -a** because the driver automatically loads when it's allocated.)

```
slave:1# kl_util -a mydriver_reloc
Adding server with relocatable /me/mydriver/LKS/mydriver_reloc
Allocating server mydriver
Server mydriver linking /me/mydriver/LKS/mydriver_reloc against /mach
Server mydriver linking relocatable "/me/mydriver/LKS/mydriver_reloc"
into loadable "/me/mydriver/LKS/mydriver_loadable"
Server mydriver Allocated
Server mydriver loading
regs.pc = 405e158
Server mydriver download complete
Server mydriver starting up
Server mydriver Loaded
slave:2#
```

5. If the loadable object file isn't currently accessible to the master computer, copy it over to the master computer.

6. Get into the NMI mini-monitor by generating an NMI as described earlier in this chapter. The slave computer is now frozen because its kernel is stopped.

On the master computer:

7. Establish the master computer's control over the slave by entering "**kattach** *hostname*", where *hostname* is the name of the slave machine.

```
(gdb) kattach slave
Attaching program: /mach to kernel on slave.
```

8. Bring the symbol information from your loadable file into GDB. To do so, enter **add-file** followed by the full pathname of the loadable file.

```
0x408f30a in kdbg_connect ()
(gdb) add-file /me/mydriver/LKS/mydriver_loadable
(gdb)
```

If you had any trouble adding your server to GDB, make sure **kl_util -s** *servername* on the slave shows your server as "Loaded." If not, load it.

9. Finally, tell GDB where your source files are with the **dir** or **idir** command. For example:

```
(gdb) dir /me/mydriver/LKS
Source directories searched: /me/mydriver/LKS://:$cdir:$cwd
(gdb)
```

Debugging with GDB

You're now ready to set breakpoints and debug your code. If you can't set breakpoints in your server, make sure it's loaded, not just allocated, on the slave computer. Remember that your server's initialization functions have already been called, so there's no point in setting a breakpoint for them.

If the slave computer wasn't booted with the **-p** option, it's useful to set a breakpoint for **panic()**. Setting this breakpoint ensures that you'll be able to use the GDB **backtrace** command to see what caused the panic.

When you're ready to continue running the kernel, use the **cont** (continue) command in GDB.

Warning: Never use the **run** command when debugging a loadable kernel server. It causes unpredictable behavior.

```
(gdb) break panic
Breakpoint 1 at 0x40230b2
(gdb) break mydriver_signoff
Breakpoint 2 at 0x105aa40c: file mydriver_main.c:17.
(gdb) break mydriver_do_log
Breakpoint 3 at 0x105aa420: file mydriver_main.c:23.
(gdb) cont
Continuing.
```

```
Breakpoint 3, mydriver_do_log (server=0) at mydriver_main.c:23
23              printf("Calling kern_serv_log.\n");
(gdb) n
24              kern_serv_log(&instance, LOG_WARNING,
(gdb) p instance
$1 = (kern_server_t) 0x10171660
(gdb) n
27              printf("Returned from calling kern_serv_log\n");
(gdb) n
28              return KERN_SUCCESS;
(gdb) info breakpoints
    Enb   Address    Where
#1   y  0x040230b2  <panic+4>
#2   y  0x105aa40c  in mydriver_signoff at mydriver_main.c:17
#3   y  0x105aa420  in mydriver_do_log at mydriver_main.c:23
(gdb) cont
Continuing.
```

To get a "(gdb)" prompt on the master computer without hitting a breakpoint, type Control-C. If that doesn't work, try again by generating an NMI at the slave computer and entering **gdb** at the "nmi>" prompt.

Ending the Debugging Session

To remove GDB from a running kernel, follow these steps:

On the master computer:

1. If you don't have a "(gdb)" prompt on the master computer, get one by typing Control-C. If that doesn't work, try again by generating an NMI at the slave computer and entering **gdb** at the "nmi>" prompt.

2. Delete all the breakpoints you've set, detach the debugger from the slave computer's kernel, and quit GDB.

```
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) detach
Detaching program: /mach pid -1
(gdb) quit
master:3#
```

10 *Kernel Support Functions*

This chapter gives detailed descriptions of the C functions provided by the NeXT Mach kernel for loadable kernel servers. Also included are some macros that behave like functions. For this chapter, the functions and macros are divided into two groups: general functions and network functions.

Network functions are those that are specifically for network modules. All the other functions are under the "General Functions" section.

Note: All functions described in this chapter work only in the kernel. The few that have user-level equivalents are noted.

Within each section, functions are subgrouped with other functions that perform related tasks. These subgroups are described in alphabetical order by the name of the first function listed in the subgroup. Functions within subgroups are also listed alphabetically, with a pointer to the subgroup's description.

For convenience, these functions are summarized in Appendix C, "Summary of Kernel Support Functions." The summary lists functions by the same subgroups used in this chapter and combines several related subgroups under a heading such as "Time Functions" or "Memory Functions." For each function, the appendix shows the calling sequence.

ASSERT()

SUMMARY Panic if an assumption isn't true

SYNOPSIS **#import <kernserv/prototypes.h>**

void **ASSERT**(int *expression*)

ARGUMENTS *expression*: A C expression that's 0 when the assumption isn't true.

DESCRIPTION **ASSERT()** is a macro that works only if you specify the DEBUG C preprocessor macro when you compile your server. If *expression* is 0, **ASSERT()** calls **panic()** after printing the line and file that the assertion failed in.

EXAMPLE In your makefile:

```
CFLAGS  = ... -DDEBUG
```

In your server:

```
ASSERT(ptr != NULL);
```

SEE ALSO **panic(), kern_serv_panic()**

assert_wait()

SUMMARY Arrange for a thread to sleep on an event

SYNOPSIS **#import <kernserv/prototypes.h>**

void **assert_wait**(int *event*, boolean_t *interruptible*)

ARGUMENTS *event*: An integer that identifies the event. Typically, this is the address of a structure. If *event* is zero, then **thread_wakeup()** won't work on the thread; only **clear_wait()** and **thread_set_timeout()** will be able to wake it up.

interruptible: Used by **clear_wait()**. If *interruptible* is false and the *interrupt_only* argument to a later call to **clear_wait()** is true, then this thread won't be waked up by that call to **clear_wait()**.

DESCRIPTION Use this function before calling **thread_block()**. This function sets up the event that the thread wants to wait for, but the thread doesn't start sleeping until it executes **thread_block()**.

EXAMPLE

```
extern  hz;

assert_wait(0, FALSE);
thread_set_timeout(hz*2);
thread_block();
```

SEE ALSO **clear_wait()**, **thread_block()**, **thread_set_timeout()**, **thread_wakeup()**, **biowait()**

bcopy()

SUMMARY Copy data into a buffer

SYNOPSIS **#import <kernserv/prototypes.h>**

void **bcopy**(void **from*, void **to*, int *length*)

ARGUMENTS *from*: Start of buffer to be copied from.

to: Start of buffer to be copied to.

length: Number of bytes to copy.

DESCRIPTION Like the C library **bcopy()** function, this function copies bytes from one buffer to another buffer in the same virtual space. This function can't be used to copy data between user space and kernel space. The caller of this function must have already checked the access rights to this memory and wired it down.

Important: Use **bytecopy()** instead of **bcopy()** if you're copying to or from hardware device space that's only 8 or 16 bits wide. (**bcopy()** often uses 32-bit accesses for efficiency, but the 68040 processor doesn't allow 32-bit accesses to 8-bit or 16-bit hardware.)

SEE ALSO **bytecopy()**, **strcpy()**, **copyin()**, **copyout()**

biodone()

SUMMARY Wake up the function doing a **biowait()** on a buffer

SYNOPSIS **#import <kernserv/prototypes.h>**

void **biodone**(struct buf **bp*)

ARGUMENTS *bp*: The address of a **buf** structure. This structure is defined in the header file **sys/buf.h**.

DESCRIPTION This function marks the buffer as done and wakes up any threads waiting for it. If the B_DONE flag is already set, **biodone()** panics. Otherwise, if B_CALL is set, **biodone()** clears it and calls the function pointed to by **bp*–>**b_iodone**. Next, if B_ASYNC is set, **biodone()** releases the buffer pointed to by *bp*; if B_ASYNC isn't set, **biodone()** clears the B_WANTED flag and wakes up all threads that had called **biowait()** on *bp*.

EXAMPLE

```
one_thread(void)
{
    struct buf  mybuf;
    . . .
    biowait (&mybuf);
    . . .
}

other_thread(struct buf *bp)
{
    . . .
    biodone(bp)
   . . .
}
```

SEE ALSO **biowait()**

---

biowait()

SUMMARY Wait until a function calls **biodone()** on a buffer

SYNOPSIS **#import <kernserv/prototypes.h>**

void **biowait**(struct buf **bp*)

ARGUMENTS *bp*: The address of a **buf** structure. This structure is defined in the header file **sys/buf.h**.

DESCRIPTION If the B_DONE flag in the buffer pointed to by *bp* is already set, this function won't sleep. Otherwise, this function sleeps until another thread calls **biodone()** on *bp*.

EXAMPLE

```
one_thread(void)
{
    struct buf  mybuf;
    . . .
    biowait (&mybuf);
    . . .
}

other_thread(struct buf *bp)
{
    . . .
    biodone(bp)
   . . .
}
```

SEE ALSO **biodone(), assert_wait()**

---

bytecopy()

SUMMARY Copy bytes into a buffer

SYNOPSIS void **bytecopy**(void **from*, void **to*, int *length*)

ARGUMENTS *from*: Start of buffer to be copied from.

to: Start of buffer to be copied to.

length: Number of bytes to copy.

DESCRIPTION This function is like **bcopy()**, except that it uses only 8-bit instructions to copy data. The **bytecopy()** function, like **bcopy()** and the C library **bcopy()** function, copies bytes from one buffer to another buffer in the same virtual space. The **bytecopy()** function can't be used to copy data between user space and kernel space. The caller of this function must have already checked the access rights to this memory and wired it down.

Note: This function is less efficient than **bcopy()**, so you should use **bcopy()** unless you're copying to or from hardware device space that's only 8 or 16 bits wide.

SEE ALSO **bcopy(), strcpy(), copyin(), copyout()**

bzero()

SUMMARY Zero out a region of memory

SYNOPSIS **#import <kernserv/prototypes.h>**

void **bzero**(void ***address*, int *length*)

ARGUMENTS *address*: The address of the first byte of the region of memory.

length: The number of bytes to write zeros to.

DESCRIPTION This acts the same as the **bzero()** C library function.

SEE ALSO **bzero(3)** UNIX manual page

clear_wait()

SUMMARY Stop a thread from waiting for an event

SYNOPSIS **#import <kernserv/prototypes.h>**
#import <kernserv/sched_prim.h>

void **clear_wait**(thread_t *thread*, int *result*, boolean_t *interrupt_only*)

ARGUMENTS *thread*: The thread to wake up.

result: The wakeup result the thread should see.

interrupt_only: If true, don't wake up the thread unless **assert_wait()** was called with *interruptible* set to true.

DESCRIPTION Use this function to wake up a thread that's waiting for an event (as the result of **assert_wait()** and **thread_block()**), whether or not the event has happened. If *interrupt_only* is false or if **assert_wait()** was called with *interruptible* set to false, then the thread is guaranteed to wake up. The thread will receive *result* when it calls **thread_wait_result()**.

EXAMPLE

```
void        new_thread(void);
extern      hz;
char        data;
thread_t    thread1;
. . .
{
    . . .
    thread1 = (thread_t)current_thread();
    kernel_thread(current_task(), new_thread);
    assert_wait(&data, FALSE);
    thread_block();
    printf("Wait result:  %d\n", thread_wait_result());
}

void new_thread()
{
    . . .
    clear_wait(thread1, THREAD_AWAKENED, FALSE);
    . . .
} /* new_thread */
```

SEE ALSO **assert_wait(), thread_block(), thread_wait_result(), thread_wakeup(), us_untimeout()**

---

clock_attributes()

SUMMARY Get information about a clock

SYNOPSIS **#import <kernserv/clock_timer.h>**

chrono_attributes_t **clock_attributes**(clock_types_t *which_clock*)

ARGUMENTS *which_clock*: Either **Calendar** or **System**.

DESCRIPTION This function lets you get information about the system and calendar clocks, such as what their accuracy is and what their maximum value is.

EXAMPLE

```
char *s;
chrono_attributes_t    attr;

if (which_clock == System)
    s = SYSTEMSTRING;
else
    s = CALENDARSTRING;
attr = clock_attributes(which_clock);
printf("%s clock has a max value of %d:%d ns and an accuracy of %d ns\n",
    s,
    *((int*) &attr->max_value), *((int *) &attr->max_value + 1),
    *((int *) &attr->resolution + 1));
```

Note that the kernel version of **printf()** does not interpret **unsigned** integers or **long long** integers. This means that printing an **ns_time_t** value results in numbers that are difficult to interpret and sometimes negative, as shown in the following printout. If you need to print time values, you can perform one of the conversions shown in the example for **clock_value()**, later in this chapter.

System clock has a max value of -1:-1 ns and an accuracy of 1000 ns.

SEE ALSO **clock_value(), ns_time_to_timeval(), set_clock()**

---

clock_value()

SUMMARY Get the current time

SYNOPSIS **#import <kernserv/clock_timer.h>**

ns_time_t **clock_value**(clock_types_t *which_clock*)

ARGUMENTS *which_clock*: Either **Calendar** or **System**.

DESCRIPTION This function returns the value of either the system clock (which starts over when the machine is booted) or the calendar clock (which continues to keep time between reboots), depending on the value of *which_clock*.

EXAMPLE

```
unsigned int    ms_time;
struct timeval  tv_time;
ns_time_t       now;

now = clock_value(System);
printf("Time since boot:  %d:%d ns == ",
    *((int*) &now), *((int *) &now + 1));

/* Since the value printed above is fairly useless, convert it */
ms_time = now / 1000000ULL;    /* convert to millisecs */
printf("%u ms == ", ms_time);
ns_time_to_timeval(now, &tv_time);
printf("%u seconds and %d microseconds.\n", tv_time.tv_sec,
    tv_time.tv_usec);
```

A typical printout:

```
Time since boot:  8942:-283072128 ns == 38409609 ms == 38409 seconds
and 609456 microseconds.
```

SEE ALSO **clock_attributes(), set_clock()**

---

copyin()

SUMMARY Copy bytes from user to kernel space

SYNOPSIS **#import <kernserv/prototypes.h>**

int **copyin**(void **from*, void **to*, int *length*)

ARGUMENTS *from*: The start of the region in user space.

to: The start of the region in kernel space.

length: The number of bytes to copy from user to kernel space.

DESCRIPTION This function works only in UNIX-style servers. It returns 0 if successful, –1 otherwise.

SEE ALSO **bcopy(), copyout()**

copyout()

SUMMARY Copy bytes from kernel to user space

SYNOPSIS **#import <kernserv/prototypes.h>**

int **copyout**(void **from*, void **to*, int *length*)

ARGUMENTS *from*: The start of the region in kernel space.

to: The start of the region in user space.

length: The number of bytes to copy from kernel to user space.

DESCRIPTION The same as **copyin()**, except the direction of the copy is reversed.

SEE ALSO **bcopy()**, **copyin()**

---

curipl()

SUMMARY Get the current interrupt level

SYNOPSIS int **curipl**(void)

DESCRIPTION This function returns the CPU interrupt level, which is a number between 0 and 7.

EXAMPLE

```
#define panic(s) (curipl() == 0 ? \
    kern_serv_panic((kern_serv_bootstrap_port(&instance), s) \
    : printf("Can't panic: %s\n", s))
```

SEE ALSO **spl*n*()**, **splx()**

current_task()

SUMMARY Get the current task

SYNOPSIS **#import <kernserv/prototypes.h>**

task_t **current_task**(void)

DESCRIPTION This macro returns the task structure for the current task. Use **current_task()** whenever you need to refer to the task in which your loadable kernel server executes. Don't use **current_task()** to refer to memory unless you specifically want the task's native memory map, and not the kernel map that your server uses.

EXAMPLE

```
kernel_thread(current_task(), new_thread);
```

SEE ALSO **kernel_thread()**

DELAY()

SUMMARY Busy-wait for a certain number of microseconds

SYNOPSIS **#import <machine/machparam.h>**

void **DELAY**(unsigned int *usecs*)

ARGUMENTS *usecs*: The number of microseconds to delay.

DESCRIPTION This macro makes the processor loop for the number of microseconds specified in the argument. Interrupts are not disabled by this function, so surround **DELAY()** with **spl*n*()** and **splx()** if interrupts need to be disabled. Because the microsecond resolution clock is used to count the spin interval, the delay is independent of CPU instruction clock speed.

This macro doesn't sleep, so it's safe to use in interrupt handlers. It's often used to wait for the hardware.

EXAMPLE

```
/* set the hardware register for at least 100 microseconds */
hardware_register = 1;
DELAY(100);
hardware_register = 0;
```

SEE ALSO **us_timeout(), us_abstimeout(), us_untimeout(), microtime(), microboot(), spl*n*(), splx()**

---

install_polled_intr()

SUMMARY Install an interrupt handler for a polled device

SYNOPSIS **#import <kernserv/prototypes.h>**

int **install_polled_intr**(int *which*, int (**my_intr*)(void))

ARGUMENTS *which:* Specifies the device and interrupt level. For devices attached through the NeXTbus interface, this should be the constant I_BUS.

my_intr: The function in your server that handles this interrupt.

DESCRIPTION This function installs an interrupt handler; you can later remove this interrupt handler by calling **uninstall_polled_intr()**.

This function returns 0 if the call is successful, or –1 if the interrupt level specified by *which* isn't capable of interrupt polling.

EXAMPLE

```
device_interrupt(void) {
    if (interrupt_is_for_us) {
        /* -process interrupt- */
        return (1);  /* say interrupt was for us */
    }
    else
        return (0);  /* it must be for someone else */
}

device_initialize(void) {
    install_polled_intr(I_BUS, device_interrupt);
    . . .
}
```

SEE ALSO **uninstall_polled_intr()**

kalloc()

SUMMARY Allocate wired-down kernel memory

SYNOPSIS **#import <kernserv/prototypes.h>**

void ***kalloc**(int *size*)

ARGUMENTS *size*: The size in bytes to be allocated.

DESCRIPTION This function is guaranteed to return wired-down memory of the requested size. The returned memory might not contain all zeros. You can't call **kalloc()** from an interrupt handler because it might sleep.

Memory returned isn't guaranteed to be aligned in any way unless size is a multiple of the page size (in which case the memory is page-aligned). If you need to ensure alignment, you should allocate twice what you need and align the address you start with to the boundary you want. Memory isn't guaranteed to be contained on the same physical page unless you allocate in multiples of the page size and keep track of the page location of addresses you use. The page size is dynamic; there's currently no way to get its value from inside the kernel. However, on 680x0-based machines, 8192 is guaranteed to be an integer multiple of the page size in bytes.

EXAMPLE

```
my_data_t *arg;

arg = (my_data_t *)kalloc(sizeof (my_data_t));
. . .
kfree(arg, sizeof (my_data_t));
```

SEE ALSO **kfree(), kget()**

---

kern_serv_bootstrap_port()

SUMMARY Get the port used to initialize your server

SYNOPSIS **#import <kernserv/kern_server_types.h>**

port_t **kern_serv_bootstrap_port**(kern_server_t **ksp*)

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

DESCRIPTION This function returns the port that the kernel uses to initialize (or "bootstrap") your server when loading it. Normally, the only reason to use this port is as an argument to **kern_serv_panic()**.

EXAMPLE

```
bootstrap_port=kern_serv_bootstrap_port(&instance);
kern_serv_panic(bootstrap_port, "Couldn't send message");
```

SEE ALSO **kern_serv_panic()**, **kern_serv_local_port()**, **kern_serv_notify_port()**, **kern_serv_port_set()**

---

kern_serv_callout()

SUMMARY Run a function in the loadable kernel server's main thread

SYNOPSIS **#import <kernserv/kern_server_types.h>**

kern_return_t **kern_serv_callout**(kern_server_t **ksp*, void (**func*)(void *), void **arg*)

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

func: The function to be called.

arg: The argument to be passed to *func*.

DESCRIPTION This function provides a way for interrupt handlers to call functions in the same loadable kernel server that may sleep or deal with a user context. The function *func* is called with argument *arg* at some point in the future.

EXAMPLE

```
void mydriver_func(mydriver_data_t data)
{

}

kern_serv_callout ((kern_server_t *)&instance, mydriver_func, (void *)arg);
```

RETURN KERN_SUCCESS: The callout was scheduled successfully.

KERN_RESOURCE_SHORTAGE: The callout couldn't be scheduled.

---

kern_serv_kernel_task_port()

SUMMARY Get the kernel's task port

SYNOPSIS **#import <kernserv/kern_server_types.h>**

port_t **kern_serv_kernel_task_port**(void)

DESCRIPTION This function returns the kernel's task port. You need to specify this port when copying out-of-line data to or from a message, as shown in the following example.

EXAMPLE

```
log_data_t tmp, local_data;
local_data = (log_data_t)kalloc(8192);

printf("Calling vm_write\n");
r = vm_write((vm_task_t)kernel_task, (vm_address_t)local_data,
    (pointer_t)log_data, 8192);
if (r != KERN_SUCCESS)
    printf("Call to vm_write failed \n");
else {
    tmp = (log_data_t)kalloc(length+1);
    (void)strncpy(tmp, local_data, length);
    printf("Contents of data are: %s\n", tmp);
}
```

---

kern_serv_local_port()

SUMMARY Determine on which port the kernel just received a message

SYNOPSIS **#import <kernserv/kern_server_types.h>**

port_t **kern_serv_local_port**(kern_server_t **ksp*)

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

DESCRIPTION This function returns the port on which the kernel just received a message in your server's behalf. The only time this function is useful is when your server was just loaded as the result of a message to one of its ports.

EXAMPLE

```
port=kern_serv_local_port(&instance);
if (port==debug_port)
    debug=TRUE;
```

SEE ALSO **kern_serv_notify_port()**, **kern_serv_port_set()**

---

kern_serv_log()

SUMMARY Put a message in the loadable kernel server's error log

SYNOPSIS **#import <kernserv/kern_server_types.h>**

void **kern_serv_log**(kern_server_t **ksp*, int *log_level*, char **format*, *arg1*, ..., *arg5*)

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

log_level: A number indicating the urgency of this log entry. Higher numbers indicate greater urgency, but the particular range of numbers used in a loadable kernel server is up to the writer of that server.

format: A string containing formatting information. See **printf()**.

arg1, ..., *arg5*: Arguments to be printed. (If you don't specify all five arguments, the compiler will display a warning, but the call will still succeed.) See **printf()**.

DESCRIPTION This function puts a message in the error log. The message can be retrieved by a user process that calls **kern_loader_get_log()**, or by the command **kl_log**.

EXAMPLE

```
kern_serv_log(&instance, 5, "Reset value of timeout to %d\n", time, 0, 0,
    0, 0);
```

SEE ALSO **log()**, **printf()**

kern_serv_notify()

SUMMARY Ask to receive notification messages about a certain port

SYNOPSIS **#import <kernserv/kern_server_types.h>**

kern_return_t **kern_serv_notify**(kern_server_t **ksp*, port_t *reply_port*,
port_t *request_port*)

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

reply_port: The port that should receive the notification messages. This should normally be the value returned by **kern_serv_notify_port()**.

request_port: The port you want to be notified about.

DESCRIPTION This function requests that notification messages about *request_port* be sent to *reply_port*. The types of notification messages are defined in the header file **sys/notify.h**.

EXAMPLE
```
notify_port=kern_serv_notify_port(&instance);
kern_serv_notify(&instance, notify_port, bootstrap_port);
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_FAILURE: The same *reply_port-request_port* pair has already been entered.

SEE ALSO **kern_serv_notify_port()**

kern_serv_notify_port()

SUMMARY Get the notify port of this server

SYNOPSIS **#import <kernserv/kern_server_types.h>**

port_t **kern_serv_notify_port**(kern_server_t **ksp*)

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

DESCRIPTION This function returns this server's notify port, which can be used in calls to **kern_serv_notify()**.

EXAMPLE
```
notify_port=kern_serv_notify_port(&instance);
kern_serv_notify(&instance, notify_port, bootstrap_port);
```

SEE ALSO **kern_serv_notify()**

---

kern_serv_panic()

SUMMARY Unload this server without panicking the system

SYNOPSIS **#import <kernserv/kern_server_reply.h>**

kern_return_t **kern_serv_panic**(port_t *bootstrap_port*, panic_msg_t *message*)

ARGUMENTS *bootstrap_port*: This server's bootstrap port, which is returned by **kern_serv_bootstrap_port()**.

message: A string to be added to the panic message that's logged.

DESCRIPTION This function unloads the server after logging a message in the kernel-server loader's log. The message is logged at the priority LOG_WARNING and contains the name of the server that called this function, followed by *message*.

This function should not be called when the CPU interrupt level is greater than 0.

This function can return, so your server should avoid doing further work after calling it. After **kern_serv_panic()** is called, the kernel attempts to call the server's unload functions.

EXAMPLE
```
kern_serv_panic(bootstrap_port,
    "my_server_main: received bad return from msg_receive");
```

RETURN KERN_SUCCESS: The server will be unloaded.

SEE ALSO **ASSERT()**, **panic()**, **kern_serv_bootstrap_port()**

kern_serv_port_gone()

SUMMARY Notify the kernel that a port will be deleted

SYNOPSIS **#import <kernserv/kern_server_types.h>**

void **kern_serv_port_gone**(kern_server_t **ksp*, port_name_t *port*)

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

port: The port that will be deleted.

DESCRIPTION Use this function to make sure that the kernel won't send any more messages to a certain port.

EXAMPLE

```
/*
 * Deallocate transmit port.
 */
kern_serv_port_gone(&instance, my_dev->xmit_port);
(void)port_deallocate((task_t)task_self(), my_dev->xmit_port);
my_dev->xmit_port = PORT_NULL;
```

SEE ALSO **kern_serv_port_proc()**, **kern_serv_port_serv()**

kern_serv_port_proc()

SUMMARY Set which function is a port's handler

SYNOPSIS **#import <kernserv/kern_server_types.h>**

kern_return_t **kern_serv_port_proc**(kern_server_t **ksp*, port_all_t *port*, port_map_proc_t *function*, int *arg*)

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

port: The port that the function should be associated with.

function: The function that handles messages sent to *port*.

arg: An integer to be passed in the call to *function* whenever *port* receives a message.

DESCRIPTION Use this function to register a message-receiving function in a handler-style (not server-style) loadable kernel server. This function provides the functionality of the HMAP load command to your server.

EXAMPLE

```
/* Create the port. */
r = port_allocate((task_t)task_self(), &port_name);
if (r != KERN_SUCCESS)
    kern_serv_panic(&instance, "couldn't allocate a port");
else printf("Created port %d\n", port_name);

/* Specify which function is its handler. */
r = kern_serv_port_proc(&instance, port_name,
    (port_map_proc_t)myhandler, 0);
if (r != KERN_SUCCESS) {
    kern_serv_panic("port_allocate failed (%d)\n", r);
    exit(1);
}

/* . . . */

kern_serv_port_gone(&instance, port_name);
port_deallocate((task_t)task_self(), port_name);
port_name = PORT_NULL;
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_RESOURCE_SHORTAGE: No more port-to-function mappings are available for your loadable kernel server.

KERN_NOT_RECEIVER: You don't have receive rights for *port*.

KERN_INVALID_ARGUMENT: *port* isn't a valid port.

SEE ALSO **kern_serv_port_gone(), kern_serv_port_serv()**

kern_serv_port_serv()

SUMMARY Set which function is a port's message server

SYNOPSIS **#import <kernserv/kern_server_types.h>**

kern_return_t **kern_serv_port_serv**(kern_server_t **ksp*, port_all_t *port*,
port_map_proc_t *function*, int *arg*)

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

port: The port that the function should be associated with.

function: The function that handles messages sent to *port*.

arg: An integer to be passed in the call to *function* whenever *port* receives a message.

DESCRIPTION This function is just like **kern_serv_port_proc()** except that it registers a function with a server-style, as opposed to a handler-style, interface. This function performs the same function as the SMAP load command.

EXAMPLE

```
/* Create the port. */
r = port_allocate((task_t)task_self(), &port_name);
if (r != KERN_SUCCESS)
    kern_serv_panic(&instance, "couldn't allocate a port");
else printf("Created port %d\n", port_name);

/* Specify which function is its server. */
r = kern_serv_port_serv(&instance, port_name, (port_map_proc_t)myserv, 0);
if (r != KERN_SUCCESS) {
    kern_serv_panic("port_allocate failed (%d)\n", r);
    exit(1);
}

/* . . . */

kern_serv_port_gone(&instance, port_name);
port_deallocate((task_t)task_self(), port_name);
port_name = PORT_NULL;
```

RETURN KERN_SUCCESS: The call succeeded.

KERN_RESOURCE_SHORTAGE: No more port-to-function mappings are available for your loadable kernel server.

KERN_NOT_RECEIVER: You don't have receive rights for *port*.

KERN_INVALID_ARGUMENT: *port* isn't a valid port.

SEE ALSO **kern_serv_port_gone()**, **kern_serv_port_proc()**

---

kern_serv_port_set()

SUMMARY Get the port set

SYNOPSIS **#import <kernserv/kern_server_types.h>**

port_set_name_t **kern_serv_port_set**(kern_server_t **ksp*)

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

DESCRIPTION This function returns the name of the port set on which messages to the loadable kernel server arrive. The kernel listens to this port set on behalf of your server. Usually, this function is used after you've temporarily removed a port from the port set, and you need the name of the port set as an argument to **port_set_add()** so you can put the port back into the port set.

EXAMPLE

```
/* Don't accept any more requests until we get rid of the old ones. */
port_set_remove((task_t)task_self(), dev->xmit_port);

. . . /* Get rid of some old requests. */

/* Re-enable listening on the port. */
port_set_add((task_t)task_self, kern_serv_port_set(&instance),
    dev->xmit_port);
```

SEE ALSO **kern_serv_port_gone()**, **kern_serv_port_proc()**, **kern_serv_port_serv()**

kern_serv_unwire_range()

SUMMARY Unwire the specified range of memory in the kernel map

SYNOPSIS **#import <kernserv/kern_server_types.h>**

kern_return_t **kern_serv_unwire_range**(kern_server_t **ksp*, vm_address_t *address*, vm_size_t *size*)

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

address: A virtual address in the kernel map.

size: The size in bytes to be wired down.

DESCRIPTION This function makes a region of kernel memory subject to swapping. Usually, you call it when you're preparing to deallocate the memory with **vm_deallocate()**.

RETURN KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: The range of memory wasn't wired down.

SEE ALSO **kalloc()**, **kfree()**, **kget()**, **kern_serv_wire_range()**

---

kern_serv_wire_range()

SUMMARY Wire down the specified range of memory in the kernel map

SYNOPSIS **#import <kernserv/kern_server_types.h>**

kern_return_t **kern_serv_wire_range**(kern_server_t **ksp*, vm_address_t *address*, vm_size_t *size*)

ARGUMENTS *ksp*: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

address: A virtual address in the kernel map.

size: The size in bytes to be wired down.

DESCRIPTION This function wires down a range of kernel memory. Usually you call it after you've copied out-of-line data into the kernel map.

RETURN KERN_SUCCESS: The call succeeded.

SEE ALSO **kalloc(), kget(), kern_serv_unwire_range()**

---

kernel_thread()

SUMMARY Start a new kernel thread in the specified task

SYNOPSIS **#import <kernserv/prototypes.h>**

thread_t **kernel_thread**(task_t *task*, void (**start*)(void))

ARGUMENTS *task*: For loadable kernel servers, this must be **current_task()**.

start: The first function to be called by the new thread.

DESCRIPTION This function can sleep, so don't call it from an interrupt handler. The new thread uses the kernel address map, but the loadable kernel server's task.

EXAMPLE

```
void new_thread(void);

kernel_thread(current_task(), new_thread);

void new_thread(void)
{
    /* Do something, then (if necessary) shut down */
    thread_terminate((thread_t)thread_self());
    thread_halt_self();
} /* new_thread */
```

SEE ALSO **current_task()**

kfree()

SUMMARY Free memory that was allocated using **kalloc()** or **kget()**

SYNOPSIS **#import <kernserv/prototypes.h>**

void **kfree**(void **address*, int *size*)

ARGUMENTS *address*: The memory to be freed. This must be exactly the same address as was returned by **kalloc()** or **kget()**.

size: The size in bytes to be freed. This must be the same size as was specified in the call to **kalloc()** or **kget()**.

DESCRIPTION The memory freed will be available for subsequent **kalloc()** and **kget()** calls only if the size they specify is the same as *size*.

EXAMPLE

```
my_data_t *arg;

arg = (my_data_t *)kalloc(sizeof (my_data_t));
. . .
kfree(arg, sizeof (my_data_t));
```

SEE ALSO **kalloc()**, **kget()**

kget()

SUMMARY Try to quickly allocate wired-down kernel memory

SYNOPSIS **#import <kernserv/prototypes.h>**

void ***kget**(int *size*)

ARGUMENTS *size*: The size in bytes to be allocated. This size, rounded up to the nearest power of 2, must be less than the page size (default 8192 bytes), or the kernel will panic.

DESCRIPTION Use this function in interrupt handlers to try to get kernel memory. If no memory of the appropriate size can be allocated without blocking, **kget()** returns 0. Otherwise, it returns the address of the chunk of memory.

EXAMPLE

```
my_data_t *arg;

arg = (my_data_t *)kget(sizeof (my_data_t));
if (arg != 0)
{ . . .
    kfree(arg, sizeof (my_data_t));
}
```

SEE ALSO **kalloc()**, **kfree()**

---

lock_alloc(), lock_free()

SUMMARY Create or destroy a lock

SYNOPSIS **#import <kernserv/prototypes.h>**

lock_t **lock_alloc**(void)
void **lock_free**(lock_t *lock*)

ARGUMENTS *lock*: The lock to be freed.

DESCRIPTION The function **lock_alloc()** returns a pointer to a new lock. Before you use the lock, you should initialize it by calling **lock_init()**.

The function **lock_free()** frees the lock structure pointed to by *lock*.

See **lock_done()** for information on using locks.

SEE ALSO **lock_done()**, **lock_init()**, **lock_read()**, **lock_write()**, **simple_lock_alloc()**, **simple_lock_free()**

lock_done()

SUMMARY Release a read or write lock

SYNOPSIS **#import <kernserv/prototypes.h>**

void **lock_done**(lock_t *lock*)

ARGUMENTS *lock*: A pointer to the lock that the reader or writer wants to release.

DESCRIPTION The **lock_*xxx*()** functions provide reader/writer synchronization. Any number of readers can read, as long as no one has a lock for writing. A writer can get a lock only if no reader or writer locks exist. Once a writer tries to get a lock, no more readers can get the lock, and the writer gets the lock as soon as the last reader releases its lock. The writer sleeps or busy-waits until it can get a lock; you determine which it does when you initialize the lock.

Use the **lock_done()** function to relinquish a read or write lock.

EXAMPLE

```
lock_write(lock1);
/* write to the protected data */
lock_done(lock1);
```

SEE ALSO **lock_alloc()**, **lock_free()**, **lock_init()**, **lock_read()**, **lock_write()**, **simple_lock_unlock()**

lock_free() → See lock_alloc()

lock_init()

SUMMARY Initialize a lock

SYNOPSIS **#import <kernserv/prototypes.h>**

void **lock_init**(lock_t *lock*, boolean_t *can_sleep*)

ARGUMENTS *lock*: A pointer to the lock that the reader or writer wants to initialize.

can_sleep: If true, threads waiting to acquire a lock can sleep. If false, threads will busy-wait while trying to acquire a lock. This should usually be true.

DESCRIPTION Use this function to initialize a lock when you first create it. See **lock_done()** for a description of how locking works. Use **lock_alloc()** to create the lock.

EXAMPLE

```
lock_t lock1 = lock_alloc();
lock_init(lock1, TRUE);

/*. . .*/

lock_free(lock1);
```

SEE ALSO **lock_alloc(), lock_done(), lock_free(), lock_read(), lock_write(), simple_lock_init()**

---

lock_read()

SUMMARY Get a lock for reading

SYNOPSIS **#import <kernserv/prototypes.h>**

void **lock_read**(lock_t *lock*)

ARGUMENTS *lock*: A pointer to the lock that the reader wants to get.

DESCRIPTION Use this function to get a lock for reading some data. If a writer holds or is waiting for a lock, you won't get the lock until the writer is done. Otherwise, you'll get the lock, even if other readers have it locked.

EXAMPLE

```
lock_t lock1 = lock_alloc();

lock_init(lock1, TRUE);
.
.
.
lock_read(lock1);
if (DONE_READING)
    lock_done(lock1);
```

SEE ALSO **lock_alloc(), lock_done(), lock_free(), lock_init(), lock_write(), simple_lock**

lock_write()

SUMMARY Get a lock for writing

SYNOPSIS **#import <kernserv/prototypes.h>**

void **lock_write**(lock_t *lock*)

ARGUMENTS *lock*: A pointer to the lock that the writer wants to get.

DESCRIPTION Use this function to get a lock for writing some data. If another writer has or is waiting for a lock, you won't get the lock until the writer is done. If any readers have locks, you won't get the lock until every reader releases its lock.

EXAMPLE

```
lock_t lock1;

lock1 = lock_alloc();
lock_init(lock1, TRUE);
.
.
.
lock_write(lock1);
if (DONE_WRITING)
    lock_done(lock1);
```

SEE ALSO **lock_alloc(), lock_done(), lock_free(), lock_init(), lock_read(), simple_lock**

log()

SUMMARY Write a message in the system log buffer

SYNOPSIS **#import <sys/syslog.h>**
#import <kernserv/prototypes.h>

int **log**(int *level*, char **format*, *arg*, ...)

ARGUMENTS *level*: The priority of the information. These priorities are defined in the header file **sys/syslog.h**.

format: A string containing formatting information. See **printf()**.

arg, ...: Arguments to be printed. See **printf()**.

DESCRIPTION Prints the time of day and who sent the message (for loadable kernel servers, it's usually sent by Mach). This function doesn't sleep, so it can be called by interrupt functions. If no process is currently reading the system log, **log()** also writes to the console. This function always returns zero.

EXAMPLE

```
log(LOG_INFO, "My driver: device %s attached\n", device_type);
```

SEE ALSO **kern_serv_log()**; **printf()**; UNIX manual pages for **syslog()** and **syslogd**

---

map_addr()

SUMMARY Convert a physical address to a virtual address

SYNOPSIS **#import <kernserv/prototypes.h>**

caddr_t **map_addr**(caddr_t *address*, int *size*)

ARGUMENTS *address*: The physical address.

size: The number of bytes to map.

DESCRIPTION This function returns a virtual address that corresponds to *address*. At least *size* bytes of hardware addresses are mapped into virtual memory. (Currently, **map_addr()** maps in multiples of the page size, nominally 8192 bytes.)

If you aren't sure whether a hardware address is implemented, you should use **map_addr()** to get a virtual address for it, and then call **probe_rb()** on the virtual address.

EXAMPLE

```
volatile unsigned int *my_reg;

my_reg = (unsigned int *)map_addr(REG_ADDRESS, 4);
if (probe_rb (my_reg))
    *my_reg |= A_FLAG;
else
    printf("Hardware at physical address 0x%x caused bus error\n",
        REG_ADDRESS);
```

SEE ALSO **probe_rb()**

---

ns_abstimeout(), ns_timeout()

SUMMARY Schedule the execution of a function at a specific time in the future

SYNOPSIS

#import <sys/callout.h>
#import <kernserv/ns_timer.h>

void **ns_abstimeout**(func *function*, vm_address_t *arg*, ns_time_t *deadline*, int *priority*)
void **ns_timeout**(func *function*, vm_address_t *arg*, ns_time_t *time*, int *priority*)

ARGUMENTS *function*: The function to call.

arg: The argument to pass to *function*.

time: The number of nanoseconds from the time **ns_timeout()** is called to the time *function* should be called.

deadline: The time, in nanoseconds since system boot time, when *function* is to be called.

priority: The priority at which to execute *function*. This should almost always be the value CALLOUT_PRI_SOFTINT0 (defined in the header file **sys/callout.h**). Other values might not be supported by future releases.

DESCRIPTION The function **ns_timeout()** schedules the function to be executed at a time relative to the current time; **ns_abstimeout()** schedules at a time relative to when the system booted. Although these functions allow nanosecond resolution to be specified, the time is rounded up to the system clock tick interval (one microsecond on 68030-based and 68040-based NeXT computers). Your driver should not rely on *function* running at exactly the specified time, since it takes an unpredictable amount of time to interrupt the current task and start the execution of *function*.

The *priority* argument specifies how the function will be executed. The value CALLOUT_PRI_SOFTINT0 means that the function will be run from a software interrupt rather than at the interrupt level of the system clock. This prevents the function from delaying interrupts at or below the system clock level.

The function is executed only once per call to **ns_timeout()** or **ns_abstimeout()**. Use **ns_untimeout()** to unschedule the execution of the function before it has been run.

EXAMPLE

```
#define ONE_SECOND 1000000000ULL
/* . . . */
/* Schedule initial execution in one second. */
ns_timeout(every_second, (void *)0, ONE_SECOND, CALLOUT_PRI_SOFTINT0);
/* . . . */

void every_second (void *arg)
{
    /* Do something. */

    /* Reschedule execution for one second from now. */
    ns_timeout(every_second, (void *)0, ONE_SECOND, CALLOUT_PRI_SOFTINT0);
}

/* mydriver_signoff:  Called when mydriver is unloaded. */
void mydriver_signoff(void)
{
    if (ns_untimeout(every_second, (void *)0))
        printf("Unscheduled every_second.\n");
    else
        printf("every_second wasn't found.\n");

    printf("My driver unloaded\n\n");
}
```

SEE ALSO **ns_untimeout(), DELAY(), clock_value(), timeval_to_ns_time()**

ns_time_to_timeval(), timeval_to_ns_time()

SUMMARY Convert between **timeval** and **ns_time_t** time formats

SYNOPSIS **#include <kernserv/ns_timer.h>**

void **ns_time_to_timeval**(ns_time_t *ns*, struct timeval **tv*)
ns_time_t **timeval_to_ns_time**(struct timeval **tv*)

ARGUMENTS *ns*: The time in nanoseconds to be converted.

tv: The equivalent time in **struct timeval** format.

DESCRIPTION The **timeval_to_ns_time()** function converts a value from **struct timeval** format (seconds and microseconds) to **ns_time_t** (nanoseconds).

The **ns_time_to_timeval()** function does the opposite conversion—from nanoseconds to seconds and microseconds. This conversion might be useful when printing a value returned by **clock_value()**, as shown in the following example.

EXAMPLE

```
unsigned int    ms_time;
struct timeval  tv_time;
ns_time_t       now;

now = clock_value(System);
printf("Time since boot:  %d:%d ns == ",
    *((int*) &now), *((int *) &now + 1));

/* Since the value printed above is fairly useless, convert it */
ms_time = now / 1000000ULL;    /* convert to millisecs */
printf("%u ms == ", ms_time);
ns_time_to_timeval(now, &tv_time);
printf("%u seconds and %d microseconds.\n", tv_time.tv_sec,
    tv_time.tv_usec);
```

ns_timeout() → See ns_abstimeout()

ns_untimeout()

SUMMARY Unschedule a timeout

SYNOPSIS **#import <kernserv/ns_timer.h>**

boolean_t **ns_untimeout**(func *function*, vm_address_t *arg*)

ARGUMENTS *function*: The function that was to be called.

arg: The argument that was to be passed to *function*.

DESCRIPTION This function is used to unschedule a call to a function previously arranged by **ns_timeout()** or **ns_abstimeout()**. Only one instance of the *function-arg* pair is removed, so it may be necessary to call **ns_untimeout()** multiple times. The function has no effect if the *function-arg* pair isn't found or if the function is already being executed. The **ns_untimeout()** function returns true if the timeout was found and unscheduled; otherwise it returns false.

EXAMPLE

```
#define ONE_SECOND 1000000000ULL
/* . . . */
/* Schedule execution in five seconds. */
ns_timeout(call_me, (void *)0, 5ULL*ONE_SECOND, CALLOUT_PRI_SOFTINT0);
/* . . . */

void call_me (void *arg)
{
    /* do something */
}

/* mydriver_signoff:  Called when mydriver is unloaded. */
void mydriver_signoff(void)
{
    if (ns_untimeout(call_me, (void *)0))
        printf("Unscheduled call_me.\n");
    else
        printf("call_me already executed.\n");

    printf("My driver unloaded\n\n");
}
```

SEE ALSO **ns_timeout(), ns_abstimeout()**

panic()

SUMMARY Hang the system and bring up the Panic window

SYNOPSIS **#import <kernserv/prototypes.h>**

void **panic**(char **string*)

ARGUMENTS *string*: The message to be printed to the console, message log, and **/usr/adm/messages**.

DESCRIPTION Calling **panic()** brings up the Panic window (similar to the NMI mini-monitor window) and either hangs or reboots the system, depending on whether you booted with the **-p** option. See Chapter 9, "Building, Loading, and Debugging Loadable Kernel Servers," for information on the Panic window.

Instead of using **panic()**, you should use **kern_serv_panic()** when possible, since it doesn't cause the whole system to panic. However, **kern_serv_panic()** can't be called when the interrupt level is greater than 0.

EXAMPLE

```
if (curipl() == 0)
    kern_serv_panic(bootstrap_port, "Couldn't get resource");
else
    panic("mydriver:  Couldn't get resource");
```

SEE ALSO **ASSERT()**, **kern_serv_panic()**

printf()

SUMMARY Display a message on the console

SYNOPSIS **#import <kernserv/prototypes.h>**

int **printf**(char **format*, *arg*, ...)

ARGUMENTS *format*: The format string. It's just like the C library **printf()** function's format string, except that the only conversions available are **%s**, **%c**, **%x**, **%d**, and **%o**. (**%X**, **%D**, **%u**, and **%O** are recognized but are treated like **%x**, **%d**, **%d**, and **%o**, respectively.)

arg, ...: Optional arguments, to be formatted according to the *format* string.

DESCRIPTION This function is a scaled-down version of the C library **printf()** function. Output goes not only to the console, but also to the message buffer and to **/usr/adm/messages**. Since **printf()** disables interrupts while printing messages, all system activities are suspended while it writes to the console.

Although **printf()** is safe to call in interrupt handlers, its output isn't guaranteed to print on the console. The message buffer, however, should be up-to-date. You can read the message buffer using the **msg** command in the NMI mini-monitor.

printf() always returns zero.

SEE ALSO **sprintf()**, **kern_serv_log()**, **log()**

---

probe_rb()

SUMMARY Check whether an address exists

SYNOPSIS **#import <kernserv/prototypes.h>**

int **probe_rb**(void **address*)

ARGUMENTS *address*: A virtual address that refers to a physical address.

DESCRIPTION This function returns 1 if *address* refers to a valid hardware address, 0 otherwise.

EXAMPLE

```
volatile unsigned int *my_reg;

my_reg = (unsigned int *)map_addr(REG_ADDRESS, 4);
if (probe_rb (my_reg))
    *my_reg |= A_FLAG;
else
    printf("Hardware at physical address 0x%x caused bus error\n",
        REG_ADDRESS);
```

SEE ALSO **map_addr()**

selthreadcache(), selthreadclear(), selwakeup()

SUMMARY Help for handling the **select**() system call

SYNOPSIS

#import <kernserv/prototypes.h>

int **selthreadcache**(void ******_waiterPtr_**)**
void **selthreadclear**(void ******_waiterPtr_**)**
int **selwakeup**(void *****_waiter_, int _collided_**)**

ARGUMENTS *waiterPtr*: A pointer to a handle for the thread that's waiting for device activity. This handle should be initialized to 0 before calling **selthreadcache()** for the first time. After the waited-for device activity occurs and the thread handle is no longer needed, the handle should be cleared by a call to **selthreadclear()**.

waiter: A handle for a thread that's waiting for the device; this value is obtained by calling **selthreadcache()**.

collided: Should be 0 if only one thread is waiting for the device; otherwise, 1.

DESCRIPTION These functions let a UNIX-style driver handle the **select()** system call. Chapter 6 has more information and examples on how to use these functions.

The **selthreadcache()** function returns in *waiterPtr* a handle for the thread that's waiting for device activity. This function returns 0 if no other thread is waiting for the device activity; otherwise, it returns a nonzero value.

The **selthreadclear()** function clears the handle pointed to by *waiterPtr*.

If *collided* is 0, **selwakeup()** wakes up the thread represented by *waiter*. If *collided* is nonzero, **selwakeup()** wakes up all parties that are sleeping as the result of a **select()** system call. The **selwakeup()** function returns no meaningful value.

set_clock()

SUMMARY Sets the current time of the calendar clock

SYNOPSIS **#import <kernserv/clock_timer.h>**

void **set_clock**(clock_types_t *which_clock*, ns_time_t *ns*)

ARGUMENTS *which_clock*: Must be **Calendar.**

ns: The time in nanoseconds (since midnight, January 1, 1970, Greenwich Mean Time) to set the clock to.

DESCRIPTION This function lets you set the current time of the calendar clock. Because the time is normally set by the Network Time Server or by the Preferences application, you don't usually need to call **set_clock()** directly.

EXAMPLE

```
unsigned int    ms_time;
ns_time_t       now;

now = clock_value(Calendar);
ms_time = now / 1000000ULL;    /* convert to millisecs */
printf("The current calendar clock time is %d ms.\n", ms_time);

set_clock(Calendar, now + timeToAdd);

now = clock_value(Calendar);
ms_time = now / 1000000ULL;    /* convert to millisecs */
printf("The new calendar clock time is %d ms.\n", ms_time);
```

SEE ALSO **clock_attributes(), clock_value(), timeval_to_ns_time(), settimeofday(2)** UNIX manual page

simple_lock()

SUMMARY Get a simple lock

SYNOPSIS **#import <kernserv/prototypes.h>**

void **simple_lock**(simple_lock_t *lock*)

ARGUMENTS *lock*: A pointer to the simple lock.

DESCRIPTION Simple locks are simple spin-loops that implement exclusive locks. They're designed to be used when you plan to hold the lock for only a short time and/or when you can't sleep.

If someone else already has the lock, this function will busy-wait until it gets the lock.

Note: Simple locks are most useful on multiprocessor systems.

EXAMPLE

```
simple_lock_t   slock;

slock = simple_lock_alloc();
simple_lock_init(slock);
/* . . . */

/* Set a lock before manipulating a data structure */
simple_lock(slock);
mydriver->data1 = VALUE;
simple_unlock(slock);

/* . . . */
simple_lock_free(lock1);
```

SEE ALSO **simple_lock_alloc(), simple_lock_free(), simple_lock_init(), simple_unlock(), lock_read(), lock_write()**

simple_lock_alloc(), simple_lock_free()

SUMMARY Allocate or free a simple lock

SYNOPSIS **#import <kernserv/prototypes.h>**

simple_lock_t **simple_lock_alloc**(void)
void **simple_lock_free**(simple_lock_t *lock*)

ARGUMENTS *lock*: The simple lock to be freed.

DESCRIPTION The **simple_lock_alloc()** function returns a pointer to a new simple lock. Before you use the simple lock, you should initialize it by calling **simple_lock_init()**.

The **simple_lock_free()** function frees the structure pointed to by *lock*.

See **simple_lock()** for information on how to use simple locks.

EXAMPLE

```
simple_lock_t   slock;

slock = simple_lock_alloc();
simple_lock_init(slock);

/* . . . */
simple_lock_free(lock1);
```

SEE ALSO **simple_lock(), simple_lock_init(), simple_unlock(), lock_alloc(), lock_free()**

simple_lock_init()

SUMMARY Initialize a simple lock

SYNOPSIS **#import <kernserv/prototypes.h>**

void **simple_lock_init**(simple_lock_t *lock*)

ARGUMENTS *lock*: A pointer to the simple lock to be initialized.

DESCRIPTION Use this function to initialize a new simple lock. You should use **simple_lock_alloc()** to create the lock.

EXAMPLE

```
simple_lock_t   slock;

slock = simple_lock_alloc();
simple_lock_init(slock);

/* . . . */
simple_lock_free(lock1);
```

SEE ALSO **simple_lock(), simple_lock_alloc(), simple_lock_free(), simple_unlock(), lock_imit()**

simple_unlock()

SUMMARY Release a simple lock

SYNOPSIS **#import <kernserv/prototypes.h>**

void **simple_unlock**(simple_lock_t *lock*)

ARGUMENTS *lock*: A pointer to the simple lock to be released.

EXAMPLE

```
simple_lock_t   slock;

slock = simple_lock_alloc();
simple_lock_init(slock);
/* . . . */

/* Set a lock before manipulating a data structure */
simple_lock(slock);
mydriver->data1 = VALUE;
simple_unlock(slock);

/* . . . */
simple_lock_free(lock1);
```

SEE ALSO **simple_lock(), simple_lock_alloc(), simple_lock_free(), simple_lock_init(), lock_done()**

---

spl*n*()

SUMMARY Set the CPU interrupt level to *n*

SYNOPSIS

#import <kernserv/m68k/spl.h>

int **spl0**(void), **spl1**(void), **spl2**(void), **spl3**(void), **spl4**(void), **spl5**(void), **spl6**(void), **spl7**(void)

DESCRIPTION The **spl*n*()** macros set the hardware interrupt level of the CPU to level *n*. This means that devices whose hardware interrupt level is greater than *n* will be serviced immediately on an interrupt. Devices with interrupt levels equal to or less than *n* will not be serviced until the CPU interrupt level drops below the device interrupt level. The **spl0()** macro sets the CPU interrupt level to the lowest level, enabling all interrupts.

The **spl*n*()** macros return an integer suitable for use with **splx()** to reset the CPU interrupt level.

The following table shows the interrupts that occur at each hardware interrupt level. Because the NMI and power fail interrupts are always serviced, **spl6()** has the same effect as **spl7()** on NeXT computers.

Interrupt Level	Interrupts at This Level
7	NMI (non-maskable interrupt) key sequence Power-fail interrupt (non-maskable)
6	System-clock timeout interrupt All DMA-completion interrupts except video out
5	RS-422 (serial) device interrupt NeXTbus interrupts
4	DSP-device interrupt
3	Disk-device interrupt SCSI-device interrupt Laser-printer device interrupt Ethernet transmit/receive device interrupts (not DMA) Sound-out underrun or sound-in overrun Video-out DMA completion interrupt Monitor-control interrupt Keyboard or mouse event Power-on switch Network-device interrupts
2	Network-related software interrupts Software interrupt 1
1	Software clock interrupts (timeouts) Software interrupt 0

EXAMPLE

```
#define spl_NB()  spl5() /* NeXTbus interrupt level */
int s;

s = spl_NB();            /* Lock out all NeXTbus interrupts. */
/* Do something that requires that we not be interrupted. */
splx(s);                 /* Return to the previous interrupt level */
```

SEE ALSO **curipl(), splx()**

splx()

SUMMARY Reset the CPU interrupt level

SYNOPSIS **#import <kernserv/m68k/spl.h>**

void **splx**(int *priority*)

ARGUMENTS *priority*: The value returned from the previous call to **spl*n*()**.

DESCRIPTION This macro returns the hardware priority interrupt level to the level that it was before issuing the last **spl*n*()** command. You must set *priority* to the value returned from the previous call to **spl*n*()**; setting it to anything else doesn't work.

EXAMPLE

```
#define spl_NB()  spl5()  /* NeXTbus interrupt level */
int  s;

s = spl_NB();   /* Lock out all NeXTbus interrupts. */
/* Do something that requires that we not be interrupted. */
splx(s);        /* Return to the previous interrupt level */
```

SEE ALSO **curipl()**, **spl*n*()**

---

sprintf()

SUMMARY Put characters into a string

SYNOPSIS **#import <kernserv/prototypes.h>**

int **sprintf**(char **string*, char **format*, *arg*, ...)

ARGUMENTS *string*: The string that you want to put the characters in.

format: The format string. It's just like the C library **printf()** function's format string, except that the only conversions available are **%s**, **%c**, **%x**, **%d**, and **%o**. (**%X**, **%D**, **%u**, and **%O** are recognized but are treated like **%x**, **%d**, **%d**, and **%o**, respectively.)

arg, ...: Optional arguments, to be formatted according to the *format* string.

DESCRIPTION This works like the C library function **sprintf()**, except that it handles only the formats allowed by the kernel **printf()** function.

SEE ALSO **printf()**, **strcat()**, **strcpy()**

---

strcat()

SUMMARY Concatenate two strings

SYNOPSIS **#import <kernserv/prototypes.h>**

char ***strcat**(char **string1*, char **string2*)

ARGUMENTS *string1*: The string to add the second string to. It must have enough space for *string2* plus a null character.

string2: The string to copy to the end of *string1*.

DESCRIPTION This acts the same as the **strcat()** C library function. It returns a pointer to *string1*.

SEE ALSO **sprintf()**, **strcpy()**, **strlen()**

---

strcmp()

SUMMARY Compare two strings

SYNOPSIS **#import <kernserv/prototypes.h>**

int **strcmp**(char **string1*, char **string2*)

ARGUMENTS *string1*: The string to be compared to *string2*.

string2: The string being compared against.

DESCRIPTION This acts the same as the **strcmp()** C library function. It returns an integer greater than, equal to, or less than 0, depending on whether *string1* is lexicographically greater than, equal to, or less than *string2*.

SEE ALSO **strlen()**

---

strcpy()

SUMMARY Copy one string to another

SYNOPSIS **#import <kernserv/prototypes.h>**

char ***strcpy**(char **to*, char **from*)

ARGUMENTS *to*: The string to copy *from* to. It must have enough space to hold all of *from*, including the null character.

from: The string to copy to *to*.

DESCRIPTION This acts the same as the **strcpy()** C library function. It returns a pointer to *to*.

SEE ALSO **sprintf()**, **strcat()**, **strlen()**

---

strlen()

SUMMARY Get the length of a string

SYNOPSIS **#import <kernserv/prototypes.h>**

int **strlen**(char **string*)

ARGUMENTS *string*: The string you want the length of.

DESCRIPTION This acts the same as the **strlen()** C library function. It returns the number of non-null characters in *string*.

SEE ALSO **strcmp()**

---

suser()

SUMMARY Check whether the user is the superuser

SYNOPSIS **#import <kernserv/prototypes.h>**

int **suser**(void)

DESCRIPTION This function is valid only for UNIX-style servers because message-based servers don't have access to user process information. If the user is the superuser, this returns 1 and sets a flag bit indicating that the process has used superuser privileges. Otherwise, it returns 0 and sets **u.u_error** to EPERM.

---

thread_block()

SUMMARY Put the current thread to sleep

SYNOPSIS **#import <kernserv/prototypes.h>**

void **thread_block**(void)

DESCRIPTION This function blocks the current thread from execution. You must call **assert_wait()** before calling **thread_block()**. This thread can be waked up by a timeout (set using **thread_set_timeout()**), by a call to **clear_wait()**, or by a call to **thread_wakeup()**.

EXAMPLE

```
extern  hz;
. . .
splx(s);
assert_wait(0, FALSE);
thread_set_timeout(hz/2);
thread_block();
```

SEE ALSO **assert_wait(), clear_wait(), thread_set_timeout(), thread_sleep(), thread_wakeup()**

---

thread_halt_self()

SUMMARY Stop the current thread

SYNOPSIS **#import <kernserv/prototypes.h>**

void **thread_halt_self**(void)

DESCRIPTION This makes the current thread stop running. You must first call **thread_terminate()** on the current thread.

EXAMPLE

```
thread_terminate((thread_t)thread_self());
thread_halt_self();
```

SEE ALSO **thread_terminate()**

---

thread_set_timeout()

SUMMARY Set a timer before calling **thread_block()**

SYNOPSIS **#import <kernserv/prototypes.h>**

void **thread_set_timeout**(int *ticks*)

ARGUMENTS *ticks*: The number of ticks to wait for. To wait for *n* seconds, this value should be *n* multiplied by the external variable **hz**.

DESCRIPTION This function sets a timer for the current thread. If you use it, you must call it between **assert_wait()** and **thread_block()**. Use the external variable **hz** (ticks per second) to convert from seconds into ticks. The thread will be waked up in *ticks*/**hz** seconds with a value of THREAD_TIMED_OUT as its wait result (obtained by calling **thread_wait_result()**).

EXAMPLE

```
splx(s);
assert_wait(0, FALSE);
thread_set_timeout(hz*2); /* set the timer to 2 seconds */
thread_block();
```

SEE ALSO **thread_block()**, **thread_wait_result()**, **us_timeout()**, **us_abs_timeout()**

---

thread_sleep()

SUMMARY Sleep until the specified event occurs

SYNOPSIS **#import <kernserv/prototypes.h>**

void **thread_sleep**(int *event*, simple_lock_t *lock*, boolean_t *interruptible*)

ARGUMENTS *event*: The event to wait for. This should be a unique integer, such as the address of a buffer. If event is zero, then **thread_wakeup()** won't work on the thread; only **clear_wait()** and **thread_set_timeout()** will be able to wake it up.

lock: The simple lock to unlock before calling **thread_block()**.

interruptible: Used by **clear_wait()**. If *interruptible* is false and the *interrupt_only* argument to a later call to **clear_wait()** is true, then this thread won't be waked up by that call to **clear_wait()**.

DESCRIPTION This is a convenient way to sleep without manually calling **assert_wait()**. This function causes the current thread to wait until the specified event occurs. The specified lock is unlocked before releasing the CPU.

This function is equivalent to:

```
assert_wait(event, interruptible);  /* assert event */
simple_unlock(lock);                /* release the lock */
thread_block();                     /* block ourselves */
```

EXAMPLE

```
extern void     thread_wakeup();
struct timeval  tv = {1, 0};

s = splmine();
simple_lock(data.slock);
if (SOME_CONDITION) {
    /* wait */
    us_timeout(thread_wakeup, (int)&data, &tv, CALLOUT_PRI_SOFTINT0);
    thread_sleep((int)&data, data.slock, TRUE);
}
simple_unlock(data.slock);
splx(s);
```

SEE ALSO **assert_wait(), simple_unlock(), thread_block(), thread_wakeup()**

---

thread_wait_result()

SUMMARY Get the wait result of the current thread

SYNOPSIS

#import <kernserv/prototypes.h>
#import <kernserv/sched_prim.h>

int **thread_wait_result**(void)

DESCRIPTION A thread that wakes up for any reason has a result in its thread structure; **thread_wait_result()** returns this result. Possible return values are defined in the header file **kernserv/sched_prim.h** as THREAD_AWAKENED, THREAD_TIMED_OUT, THREAD_INTERRUPTED, THREAD_SHOULD_TERMINATE, and THREAD_RESTART.

EXAMPLE

```
assert_wait(&data, FALSE);
thread_block();
printf("Wait result:  %d\n", thread_wait_result());
```

SEE ALSO **thread_set_timeout(), thread_wakeup()**

thread_wakeup()

SUMMARY Wake up all threads that are waiting for the specified event

SYNOPSIS void **thread_wakeup**(int *event*)

ARGUMENTS *event*: The event that was specified in the matching **assert_wait()** or **thread_sleep()** call. This should not be zero, since this function can't wake up threads that were put to sleep with an event of zero.

DESCRIPTION The threads that this macro wakes up have THREAD_AWAKENED in their wait result (obtainable by calling **thread_wait_result()**).

Warning: This function must be called at an interrupt level of IPLSCHED or below. You can use **curipl()** to determine what the interrupt level is.

EXAMPLE

```
extern void     thread_wakeup();
struct timeval  tv = {1, 0};      /* 1-second timeout */

s = splmine();
simple_lock(data.slock);
if (SOME_CONDITION) {
    /* wait */
    us_timeout(thread_wakeup, (int)&data, &tv, CALLOUT_PRI_SOFTINT0);
    thread_sleep((int)&data, data.slock, TRUE);
}
simple_unlock(data.slock);
splx(s);
```

In an interrupt handler for a NeXTbus driver:

```
if ( (sp->flags & SERVER_THREAD_PAUSED) != 0 )
{
    kern_serv_callout( &instance, thread_wakeup,
        (void *)&sp->server_thread );
}
```

SEE ALSO **assert_wait()**, **thread_sleep()**

timeval_to_ns_time() → See ns_time_to_timeval()

---

uninstall_polled_intr()

SUMMARY Remove an interrupt handler for a polled device

SYNOPSIS **#import <kernserv/prototypes.h>**

int **uninstall_polled_intr**(int *which*, int (**my_intr*)())

ARGUMENTS *which:* Specifies the device and interrupt level. For devices attached through the NeXTbus interface, this should be the constant I_BUS, which is defined in the header file **architecture/m68k/intr.h.**

my_intr: The function in your server that handles this interrupt.

DESCRIPTION This function removes *my_intr* from the list of functions that are called when an interrupt occurs at interrupt level *which.*

This function returns 0 if the call is successful. It returns –1 if the interrupt level specified by *which* isn't capable of interrupt polling, or if *my_intr* isn't found.

EXAMPLE

```
device_cleanup()
{
    /* . . . */
    uninstall_polled_intr(I_BUS, device_interrupt);
    /* . . . */
}
```

SEE ALSO **install_polled_intr()**

htonl(), htons(), ntohl(), ntohs()

SUMMARY Convert values between host and network byte order

SYNOPSIS

#import <netinet/in.h>

u_long **htonl**(u_long *hostlong*)
u_short **htons**(u_short *hostshort*)
u_long **ntohl**(u_long *netlong*)
u_short **ntohs**(u_short *netshort*)

DESCRIPTION These functions and macros simulate the C library functions of the same name. See the UNIX manual page for **byteorder** for more information.

if_attach()

SUMMARY Initialize and install a new netif

SYNOPSIS

#import <net/netif.h>

netif_t **if_attach**(if_init_func_t *init_func*, if_input_func_t *input_func*,
if_output_func_t *output_func*, if_getbuf_func_t *getbuf_func*,
if_control_func_t *control_func*, const char **name*, unsigned int *unit*, const char **type*,
unsigned int *mtu*, unsigned int *flags*, netif_class_t *class*, void **private*)

ARGUMENTS *init_func*: The initialization function of this module.

input_func: The input function of this module.

output_func: The output function of this module.

getbuf_func: The buffer allocation function of this module.

control_func: The control function of this module.

name: A constant string that names this module (for example, "en").

unit: The unit number of this module (for example, **0**).

type: A constant string that describes the type of this module (for example, "10MB Ethernet").

mtu: The maximum transfer unit (for example, **1500** for Ethernet). This is the maximum amount of data your module can send or receive. Note that protocol-level modules must return the minimum of either the protocol limit or the network device driver's limit (minus header information).

flags: Initial flags for the interface. Possible values are:

IFF_UP:	If true, this interface is working.
IFF_BROADCAST:	If true, this interface supports broadcast.
IFF_LOOPBACK:	If true, this interface is local only.
IFF_POINTTOPOINT:	If true, this is a point-to-point interface.

class: The class of this interface. Possible values are:

NETIFCLASS_REAL:	Network driver (a module that manipulates the hardware).
NETIFCLASS_VIRTUAL:	Protocol handler (a module that doesn't talk to hardware).
NETIFCLASS_SNIFFER:	Packet sniffer (a module that merely examines packets).

private: Private data, which can be retrieved using **if_private()**.

DESCRIPTION Initializes a new netif and installs it, returning the resulting netif. Network device drivers should call this function at load time. Protocol handlers and packet sniffers should instead call **if_registervirtual()** at load time, supplying as an argument the callback function that calls **if_attach()**.

The first five arguments are the functions that are associated with netifs. These functions are described in Chapter 8, "Network Modules."

This function doesn't check any of its arguments, so it always succeeds.

EXAMPLE

```
ifp = if_attach(NULL, myhandler_input, myhandler_output,
    myhandler_getbuf, myhandler_control, name, unit, IFTYPE_IP,
    MYMTU, IFF_BROADCAST, NETIFCLASS_VIRTUAL, ifprivate);
```

SEE ALSO **if_flags(), if_output(), if_init(), if_control(), if_ioctl(), if_getbuf(), if_handle_input()**

if_class()

SUMMARY Get the class of a netif

SYNOPSIS **#import <net/netif.h>**

netif_class_t **if_class**(netif_t *netif*)

ARGUMENTS *netif*: The module to get the class for.

DESCRIPTION This function returns the class of *netif*. The value returned is one of the following:

- NETIFCLASS_REAL
- NETIFCLASS_VIRTUAL
- NETIFCLASS_SNIFFER

EXAMPLE

```
netif_t ni;

for (ni = iflist_first(); ni != NULL; ni = iflist_next(ni)) {
    printf("name: %s%d, class: %s\n", if_name(ni), if_unit(ni),
        (if_class(ni) == NETIFCLASS_REAL) ? "real" :
            ((if_class(ni) == NETIFCLASS_VIRTUAL) ? "virtual" :
          ((if_class(ni) == NETIFCLASS_SNIFFER) ? "sniffer" : "unknown")));
}
```

SEE ALSO **iflist_first(), iflist_next()**

if_collisions(), if_collisions_set()

SUMMARY Get or set the number of collisions

SYNOPSIS **#import <net/netif.h>**

unsigned int **if_collisions**(netif_t *netif*)
void **if_collisions_set**(netif_t *netif*, unsigned int *collisions*)

ARGUMENTS *netif*: The module to get or set collision data for.

collisions: The new number of collisions.

DESCRIPTION The functions **if_collisions()** and **if_collisions_set()** get and set, respectively, the number of collisions encountered. Only the module corresponding to *netif* should call **if_collisions_set()**.

EXAMPLE

```
printf("Number of collisions encountered so far:  %d\n",
     if_collisions(netif));
```

---

if_control(), if_getbuf(), if_init(), if_ioctl(), if_output()

SUMMARY Call one of the functions associated with a netif

SYNOPSIS

#import <net/netif.h>

int **if_control**(netif_t *netif*, const char **command*, void **data*)
netbuf_t **if_getbuf**(netif_t *netif*)
int **if_init**(netif_t *netif*)
int **if_ioctl**(netif_t *netif*, unsigned int *command*, void **data*)
int **if_output**(netif_t *netif*, netbuf_t *packet*, void **address*)

ARGUMENTS *netif*: The module whose function should be called.

command: The control command that should be executed.

data: Data specific to the control command.

packet: The packet that should be passed to *netif*'s output function.

address: The address that should be specified in the call to *netif*'s output function.

DESCRIPTION The **if_control()** function calls *netif*'s control function, **if_getbuf()** calls *netif*'s buffer allocation function, **if_init()** calls *netif*'s initialization function, and **if_output()** calls *netif*'s output function.

It is recommended that you don't use **if_ioctl**(); it's provided only for compatibility with UNIX code that operates using **ioctl**. The **if_ioctl**() function calls *netif*'s control function.

Except for **if_getbuf**(), these functions return ENXIO if the corresponding function isn't implemented in *netif*; otherwise they return the value returned by the call to the corresponding function. The **if_getbuf**() function returns NULL if the corresponding function isn't implemented.

EXAMPLE

```
nb = if_getbuf(ifp);
if (nb == NULL)
    return ENOBUFS;
```

SEE ALSO **if_attach**()

---

if_detach()

SUMMARY Detach a netif so it can be unloaded

SYNOPSIS **#import <net/netif.h>**

void **if_detach**(netif_t *netif*)

ARGUMENTS *netif*: The module to detach.

DESCRIPTION The **if_detach**() function removes *netif* from the system lists of attached netifs. You should call this function before unloading a network module.

EXAMPLE

```
netif_t ni;

for (ni = iflist_first(); ni != NULL; ni = iflist_next(ni)) {
    if (if_unit(ni) == myunit && if_type(ni) == mytype &&
        strcmp(if_name(ni), myname) == 0) {
            if_detach(ni);
            break;
    }
}
```

if_flags(), if_flags_set()

SUMMARY Get or set the flags associated with a netif

SYNOPSIS **#import <net/netif.h>**

unsigned int **if_flags**(netif_t *netif*)
void **if_flags_set**(netif_t *netif*, unsigned int *flags*)

ARGUMENTS *netif*: The module to get or set flags for.

flags: The new flags.

DESCRIPTION The functions **if_flags**() and **if_flags_set**() get and set, respectively, the flags associated with *netif*. Only the module corresponding to *netif* should use **if_flags_set**().

Possible flag values are:

IFF_UP:	If true, this interface is working.
IFF_BROADCAST:	If true, this interface supports broadcast.
IFF_LOOPBACK:	If true, this interface is local only.
IFF_POINTTOPOINT:	If true, this is a point-to-point interface.

EXAMPLE

```
if_flags_set(ifp, if_flags(ifp) | IFF_UP);
```

SEE ALSO **if_attach**()

if_getbuf() → See if_control()

if_handle_input()

SUMMARY Dispatch an input packet to a protocol handler

SYNOPSIS **#inıport <net/netif.h>**

int **if_handle_input**(netif_t *netif*, netbuf_t *packet*, void **extra*)

ARGUMENTS *netif*: This module, which must be a network device driver.

packet: The input packet.

extra: Any extra data that might be needed by the protocol handler.

DESCRIPTION Call this in a network device driver to have an input packet dispatched to a protocol handler. This function calls the input functions of one or more protocol handlers, passing along the *packet* and *extra* arguments.

This function returns EAFNOSUPPORT if no protocol handler accepts the packet.

EXAMPLE

```
if (nb == 0) {
    printf ("Error:  buffer is null\n");
    goto resetup;
}
else {
    if_handle_input(netif, nb, NULL);
}
```

---

if_ierrors(), if_ierrors_set(), if_oerrors(), if_oerrors_set()

SUMMARY Get or set the number of input or output errors

SYNOPSIS **#import <net/netif.h>**

unsigned int **if_ierrors**(netif_t *netif*)
void **if_ierrors_set**(netif_t *netif*, unsigned int *ierrors*)
unsigned int **if_oerrors**(netif_t *netif*)
void **if_oerrors_set**(netif_t *netif*, unsigned int *oerrors*)

ARGUMENTS *netif*: The module for which to access the number of errors.

ierrors: The number of input errors.

oerrors: The number of output errors.

DESCRIPTION The functions **if_ierrors**() and **if_ierrors_set**() get and set, respectively, the number of input errors encountered. Only the module corresponding to *netif* should call **if_ierrors_set**().

The **if_oerrors()** function and the **if_oerrors_set()** function get and set the number of output errors encountered. Again, only the *netif*'s module should call **if_oerrors_set()**.

EXAMPLE

```
error = if_output(lowernetif, nb, (void *)addr);
if (error == 0)
    if_opackets_set(netif, if_opackets(netif) + 1);
else
    if_oerrors_set(netif, if_oerrors(netif) + 1);
return (error);
```

if_init() → See if_control()

if_ioctl() → See if_control()

---

if_ipackets(), if_ipackets_set(), if_opackets(), if_opackets_set()

SUMMARY Get or set the number of packets received or sent

SYNOPSIS

#import <net/netif.h>

unsigned int **if_ipackets**(netif_t *netif*)
void **if_ipackets_set**(netif_t *netif*, unsigned int *ipackets*)
unsigned int **if_opackets**(netif_t *netif*)
void **if_opackets_set**(netif_t *netif*, unsigned int *opackets*)

ARGUMENTS

netif: The module whose packet information is to be accessed.

ipackets: The number of input packets handled by this module since it was loaded.

opackets: The number of packets sent to a lower level by this module since it was loaded.

DESCRIPTION

The **if_ipackets()** function gets and the **if_ipackets_set()** function sets the number of input packets handled. Only the module specified by *netif* should call **if_ipackets_set()**.

Similarly, **if_opackets()** gets and **if_opackets_set()** sets the number of output packets sent. Only *netif*'s module should call **if_opackets_set()**.

EXAMPLE

```
error = if_output(lowernetif, nb, (void *)addr);
if (error == 0) {
    if_opackets_set(netif, if_opackets(netif) + 1);
} else {
    if_oerrors_set(netif, if_oerrors(netif) + 1);
}
return (error);
```

---

if_mtu(), if_name(), if_private(), if_type(), if_unit()

SUMMARY Get information about a netif

SYNOPSIS

#import <net/netif.h>

unsigned int **if_mtu**(netif_t *netif*)
const char ***if_name**(netif_t *netif*)
void ***if_private**(netif_t *netif*)
const char ***if_type**(netif_t *netif*)
unsigned int **if_unit**(netif_t *netif*)

ARGUMENTS *netif*: The netif whose data is being requested.

DESCRIPTION These functions return the following information about *netif*:

if_mtu():	Its maximum transfer unit (for example, **1500** for Ethernet).
if_name():	Its name (for example, "en").
if_private():	Its private data.
if_type():	Its type string (for example, "10MB Ethernet").
if_unit():	Its unit number (for example, **0**).

See Chapter 8 for more information on netifs. Only the module specified by *netif* should call **if_private()**.

EXAMPLE

```
((venip_private_t *)if_private(netif))->lowernetif = realnetif;
```

SEE ALSO **if_flags(), if_flags_set(), if_attach()**

if_oerrors() → See if_ierrors()

if_oerrors_set() → See if_ierrors()

if_opackets() → See if_ipackets()

if_opackets_set() → See if_ipackets()

if_output() → See if_control()

if_private() → See if_mtu()

---

if_registervirtual()

SUMMARY Register a callback function that calls **if_attach()**

SYNOPSIS **#import <net/netif.h>**

void **if_registervirtual**(if_attach_func_t *attach_func*, void **private*)

ARGUMENTS *attach_func*: The callback function.

private: Data to be passed to the callback function.

DESCRIPTION For use by protocol handlers and packet sniffers. This function registers a callback function that should call **if_attach()**. See Chapter 8 for more information on implementing this callback function.

EXAMPLE

```
if_registervirtual(myhandler_attach, NULL);
```

if_type() → See if_mtu()

if_unit() → See if_mtu()

iflist_first(), iflist_next()

SUMMARY Find the first or next netif in the system netif list

SYNOPSIS

#import <net/netif.h>

netif_t **iflist_first**(void)
netif_t **iflist_next**(netif_t *netif*)

ARGUMENTS *netif*: The previous netif in the list.

DESCRIPTION These functions are useful for cycling through the system list of attached netifs. The **iflist_first()** function returns the first netif in the list. The **iflist_next()** function returns the netif after *netif* in the list.

EXAMPLE

```
netif_t ni;

for (ni = iflist_first(); ni != NULL; ni = iflist_next(ni)) {
    printf("name: %s%d, type: %s\n", if_name(ni), if_type(ni));
}
```

inet_queue()

SUMMARY Give an IP input packet to the kernel for processing

SYNOPSIS

#import <net/netif.h>
#import <net/netbuf.h>

void **inet_queue**(netif_t *netif*, netbuf_t *netbuf*)

ARGUMENTS *netif*: The protocol handler.

netbuf: The packet to hand over.

DESCRIPTION IP protocol handlers wishing to hand over their input packets to the kernel for processing should call this function.

You can safely call this function from an interrupt handler, since it doesn't block.

EXAMPLE

```
nb_shrink_top(netbuf, HDRSIZE);
if_ipackets_set(netif, if_ipackets(netif) + 1);
inet_queue(netif, netbuf);
```

---

nb_alloc(), nb_alloc_wrapper()

SUMMARY Allocate a netbuf or a netbuf wrapper

SYNOPSIS

#import <net/netbuf.h>

netbuf_t **nb_alloc**(unsigned int *size*)
netbuf_t **nb_alloc_wrapper**(void **data*, unsigned int *size*, void *freefunc*(void *),
void **freefunc_arg*)

ARGUMENTS

size: The size of the data to be stored.

data: The data to be stored.

freefunc: The function to be called when the netbuf is freed.

freefunc_arg: The argument to be passed to *freefunc*.

DESCRIPTION

The **nb_alloc**() function allocates a netbuf containing *size* bytes. It returns the netbuf.

The **nb_alloc_wrapper**() function allocates only a wrapper for a netbuf. Use this function when you have already allocated space for the packet. The returned netbuf's original start of data is *data*, and its bottom pointer is initialized to *data* + *size* −1. If **nb_free**() is called on the returned netbuf, *freefunc* will be called with the argument *freefunc_arg*. The value of *freefunc* can't be null.

Note: Don't call either **nb_alloc**() or **nb_alloc_wrapper**() from an interrupt handler. (Both functions can sleep.)

Both **nb_alloc**() and **nb_alloc_wrapper**() return a null pointer if they fail.

EXAMPLE

```
nb = nb_alloc_wrapper((void *)buf, HDR_SIZE + MYMTU,
    mydriver_buf_put, (void *)buf);
```

SEE ALSO **nb_free(), nb_free_wrapper()**

nb_free(), nb_free_wrapper()

SUMMARY Free a netbuf or only its wrapper

SYNOPSIS **#import <net/netbuf.h>**

void **nb_free**(netbuf_t *nb*)
void **nb_free_wrapper**(netbuf_t *nb*)

ARGUMENTS *nb*: The netbuf to be freed (or whose wrapper is to be freed).

DESCRIPTION The **nb_free**() function frees the netbuf *nb*. The **nb_free_wrapper**() function frees only the wrapper of *nb*, leaving its data area intact.

EXAMPLE

```
nb_free_wrapper(nb);
```

SEE ALSO **nb_alloc**(), **nb_alloc_wrapper**()

nb_grow_bot(), nb_shrink_bot(), nb_grow_top(), nb_shrink_top()

SUMMARY Change the size of a netbuf

SYNOPSIS **#import <net/netbuf.h>**

int **nb_grow_bot**(netbuf_t *nb*, unsigned int *size*)
int **nb_shrink_bot**(netbuf_t *nb*, unsigned int *size*)
int **nb_grow_top**(netbuf_t *nb*, unsigned int *size*)
int **nb_shrink_top**(netbuf_t *nb*, unsigned int *size*)

ARGUMENTS *nb*: The netbuf to be affected.

size: The number of bytes to add or delete from the top or bottom pointer.

DESCRIPTION The **nb_grow_bot**() function moves the bottom pointer down. After this call, the data is assumed to end *size* bytes after where it used to end (the data area has effectively grown).

The **nb_shrink_bot()** function moves the bottom pointer up, effectively shrinking the data area. After this call, the data is assumed to end *size* bytes before where it used to end.

The **nb_grow_top()** function moves the top pointer up, enlarging the data area. After this call, the data is assumed to start *size* bytes before where it used to start.

The **nb_shrink_top()** function moves the top pointer down, shrinking the data area. After this call, the data is assumed to start *size* bytes beyond where it used to start.

These functions perform no error checking, so they always succeed and return zero.

Warning: Writing to space outside of the original starting and ending points will cause serious errors, since the extra memory doesn't belong to the netbuf data section.

EXAMPLE

```
nb_shrink_top(netbuf, HDRSIZE);
if_ipackets_set(netif, if_ipackets(netif) + 1);
inet_queue(netif, netbuf);
```

SEE ALSO **nb_size()**

---

nb_map()

SUMMARY Get a pointer to the data stored in a netbuf

SYNOPSIS

#import <net/netbuf.h>

char ***nb_map**(netbuf_t *nb*)

ARGUMENTS *nb*: The netbuf whose data you want.

DESCRIPTION Returns a pointer to the data stored in *nb*. The pointer is valid only until another **nb_*()** function is called on *nb*.

This function returns a null pointer if it fails.

EXAMPLE

```
char *map = nb_map(nb);
```

SEE ALSO **nb_read(), nb_write(), nb_size()**

nb_read(), nb_write()

SUMMARY Access data in a netbuf

SYNOPSIS **#import <net/netbuf.h>**

int **nb_read**(netbuf_t *nb*, unsigned int *offset*, unsigned int *size*, void **target*)
int **nb_write**(netbuf_t *nb*, unsigned int *offset*, unsigned int *size*, void **source*)

ARGUMENTS *nb*: The netbuf whose data you want to read or write.

offset: The offset of the start of data from the beginning of the netbuf.

size: The number of bytes to be read or written.

target: The place to put the data.

source: The place to read the data from.

DESCRIPTION The **nb_read**() function reads data from *nb* into *target*. It starts at offset *offset* from the starting point in the data and reads *size* bytes into *target*.

The **nb_write**() function writes data into *nb*. It starts writing at offset *offset* from the starting point in the data and writes *size* bytes from *source*.

Both **nb_read**() and **nb_write**() return zero if the call was successful; otherwise, they return a nonzero value.

EXAMPLE

```
char    buf[MAXTRAILERBUF];

/*
 * Save copy of data.
 */
nb_read(nb, HDRSIZE, offset, &buf);
```

SEE ALSO **nb_map()**

nb_shrink_bot() → See nb_grow_bot()

nb_shrink_top() → See nb_grow_bot()

nb_size()

SUMMARY Get the size of the data stored in a netbuf

SYNOPSIS **#import <net/netbuf.h>**

unsigned int **nb_size**(netbuf_t *nb*)

ARGUMENTS *nb*: The netbuf to get the size of.

DESCRIPTION Returns the size (in bytes) of the data stored by *nb*.

EXAMPLE

```
if (HDRSIZE + offset + size > nb_size(nb)) {
    return (EAFNOSUPPORT);
}
```

SEE ALSO **nb_grow_bot(), nb_shrink_bot(), nb_grow_top(), nb_shrink_top(), nb_map()**

nb_write() → See nb_read()

ntohl() → See htonl()

ntohs() → See htonl()

A *Utilities for Loadable Kernel Servers*

This appendix describes the syntax of several of the commands that operate on loadable kernel servers:

- The kernel-server loader, **kern_loader**, is the program that loads servers into the kernel. It's also discussed in Chapter 5, "Overview of Loadable Kernel Servers."
- The kernel-server linker, **kl_ld**, links object files and command scripts into a relocatable object file. The format of command scripts is discussed in detail in this appendix. An example of using **kl_ld** is in Chapter 9, "Building, Loading, and Debugging Loadable Kernel Servers."
- The kernel-server utility, **kl_util**, is useful for loading and unloading servers, as well as getting information about servers. It's also discussed briefly in Chapter 9.
- The kernel-server log command, **kl_log**, lets you see log messages from a server. Log messages are discussed in Chapter 6, "Designing Loadable Kernel Servers."

Kernel-Server Loader (kern_loader)

The kernel-server loader, **kern_loader**, is the task that adds loadable kernel servers to the kernel. The kernel-server loader works by listening to the ports of known loadable kernel servers. When it intercepts a request for a loadable kernel server, it loads the server and initializes it to respond to this request and subsequent requests.

The kernel-server loader also listens on its own port for requests made through the kernel-server loader functions, a group of user-level functions. The kernel-server loader functions can be used to add and delete known servers, to load servers into the kernel and unload running servers from the kernel, and to get status information. You can use the kernel-server utility, **kl_util**, to communicate with the kernel-server loader (as described later in this appendix), or you can write your own program using the kernel-server loader functions. The kernel-server loader functions are documented in Chapter 3, "Using Loadable Kernel Servers."

When invoked, the kernel-server loader reads its configuration file, **/etc/kern_loader.conf**. This file contains a list of relocatable object files, one for each kernel server that is to be prepared for loading into the kernel. Here's a sample **kern_loader.conf** file:

```
/usr/lib/kern_loader/Midi/mididriver_reloc
/usr/local/lib/kern_loader/Mydriver/mydriver_reloc
```

Starting kern_loader

The **kern_loader** daemon is called automatically during system startup. If it's killed, you normally can't restart it because the Bootstrap Server won't let any process except **mach_init** register the "server_loader" service. However, if you change a couple of lines in the Bootstrap Server configuration file and then reboot, the Bootstrap Server will let you reinvoke **kern_loader** in the future. Specifically, you should change the following lines in **/etc/bootstrap.conf**:

```
services NetMessage . . .;
. . .
server "/usr/etc/kern_loader -n" services server_loader;
```

to the following:

```
services NetMessage . . . server_loader;
. . .
server "/usr/etc/kern_loader -n";
```

After you make that change to **/etc/bootstrap.conf** and reboot, you can reinvoke **kern_loader** at any time, as follows:

/usr/etc/kern_loader [**-d**] [**-n**] [**-v**] [*relocatable ...*]

The command-line options are:

-d	Don't detach from the invoking terminal; stay in the foreground.
-n	Don't fork another process to be **kern_loader**. This is necessary in the Bootstrap Server configuration file because the Bootstrap Server keeps track of all its servers.
-v	Display debugging information.
relocatable ...	The name of one or more relocatable object files to be read (before those listed in **/etc/kern_loader.conf**).

Kernel-Server Linker (kl_ld)

The relocatable object file of your loadable kernel server must contain certain information: the name of your server, which functions to call to initialize the server, the names of message-handling functions, the name of your server's instance variable, and so on. You put this information into the relocatable object file by using **kl_ld** to link your server.

The syntax for using **kl_ld** follows:

kl_ld -n *server_name* **-i** *instance_var* **-l** *load_cmds_file* [**-u** *unload_cmds_file*] [**-d** *loadable_name*] **-o** *output_file input_file ...*

where:

-n *server_name* — Specifies the name of the kernel server. This name is used in calls to the kernel-server loader functions (such as **kern_loader_load_server()**) and in the **kl_util** and **kl_log** command lines.

-i *instance_var* — Specifies the name of the kernel server's instance variable. This variable's structure must start with a field of type **kern_server_t** (defined in the header file **kernserv/kern_server_types.h**).

-l *load_cmds_file* — Specifies the name of the script that contains commands that **kern_loader** must execute when it loads your server. This file is read into the relocatable object file when you create it. If you want to change the load commands, you must recreate the relocatable object file.

-u *unload_cmds_file* — Specifies the name of the script that contains commands that **kern_loader** must execute when it *un*loads your server. Like load commands, unload commands are read into the relocatable object file when you create it. Thus, you must recreate the relocatable object file if you want to change the unload commands.

-d *loadable_name* — Specifies the pathname of the loadable object file that **kern_loader** creates from the relocatable object file. This pathname can be either absolute or relative to the directory containing the relocatable object file. Use this option to make **kern_loader** put the loadable object file in a place where the GNU source level debugger, GDB, can easily find and use it.

-o *output_file* — Specifies the name of the relocatable object file that is created. The **kern_loader** will later relocate this file against the kernel.

input_file ... — The object files to be linked into the relocatable object file.

The following example shows a makefile that creates a relocatable object file.

Note: On the last line of the command for the "$(NAME)_reloc" target, "$@" refers to "$(NAME)_reloc".

```
NAME=slot
OFILES= slot_server.o slot_handler.o
CFLAGS= -g -DKERNEL -DKERNEL_FEATURES -DMACH
.
.
.
$(NAME)_reloc:  $(OFILES) Load_Commands Unload_Commands
                kl_ld -n $(NAME) -i instance -l Load_Commands \
                    -u Unload_Commands -o $@ $(OFILES)

.c.o:
$(CC) $(CFLAGS) -c $*.c -o $*.o
```

Load Commands

The load commands script can have the commands described in this section. The script must have at least one of the following commands: HMAP, SMAP, or START.

ADVERTISE — Specifies the name of a port that is to be allocated and advertised with the Network Name Server. When **kern_loader** receives messages on any advertised port, the kernel server will be loaded into the kernel and initialized. As part of the initialization sequence, receive rights for the advertised port are forwarded to the kernel server. The message will then be forwarded by **kern_loader** to the loaded kernel server.

Syntax: **ADVERTISE** *port*

CALL — Specifies the name of a function to be called with the specified integer argument as part of the server initialization sequence. If the script has multiple CALL commands, they'll be executed in order.

Syntax: **CALL** *function integer*

DETACH

Specifies that the server should never be unloaded. The DETACH command makes **kern_loader** treat any request to unload the server as an error. Requests to deallocate the server will appear to succeed, but although **kern_loader** stops keeping track of it, the server will remain loaded in kernel memory. The DETACH command is necessary for the correctness of some network protocols.

HMAP

Specifies the mapping of a port to a message-handling function in the kernel server. When **kern_loader** receives a message on this port, it calls the function with the global variable or integer argument you specify. This function must have a handler interface, as opposed to a server interface (see Chapter 6, "Designing Loadable Kernel Servers"). To advertise this port with the Network Name Server, use the ADVERTISE command, as described previously in this section.

Syntax: **HMAP** *port_name handler_function argument*

PORT_DEATH

Specifies a function in the server to be called when a port death message is received on its behalf.

Syntax: **PORT_DEATH** *function_name*

SMAP

Specifies the mapping of a port to a message-handling function within the kernel server. When **kern_loader** receives a message on this port, it calls the function with the integer argument you specify. This function must have a server interface, as opposed to a handler interface (see Chapter 6). To advertise this port with the Network Name Server, use the ADVERTISE command, as described previously in this section.

Syntax: **SMAP** *port_name server_function integer*

START

Causes the kernel server to be started immediately, rather than waiting for a message to be received on one of its advertised ports. This is most appropriate for kernel servers that don't listen on any ports, or are wired into kernel data structures for nonserver-style access.

Syntax: **START**

WIRE — Causes the text and data of the loaded kernel server to be wired down (memory resident), making the kernel server immune from unexpected page faults. You must use WIRE if any part of your kernel server can be called from an interrupt handler. If you use WIRE, your kernel server is wired down before any other load commands are executed.

Syntax: **WIRE**

Here's an example of a load commands script.

```
CALL  slot_init 0

PORT_DEATH  slot_port_death

# Associate ports with proc/arg
SMAP  slot0 slot_msg 0
SMAP  slot2 slot_msg 1
SMAP  slot4 slot_msg 2
SMAP  slot6 slot_msg 3

# Server contains interrupt handler code, and so must be wired down
WIRE

# Start this server up immediately
START
```

Unload Commands

The unload commands script can have only CALL commands:

CALL — Specifies the name of a function to be called as part of server shutdown. The function will be passed the specified integer.

Syntax: **CALL** *function integer*

Here's an example of an unload commands script.

```
# Termination

CALL   slot_signoff  0
```

Kernel-Server Utility (kl_util)

The kernel-server utility **/usr/etc/kl_util** lets you communicate with the kernel-server loader. Various options allow you to query the kernel loader for the status of all registered kernel servers, load a kernel server into the kernel, and remove one or more kernel servers from the kernel.

The command-line options to **kl_util** are as follows:

-a *server_reloc_file_name ...*
: Causes **kern_loader** to allocate resources for the specified kernel server or servers. Each added server will have kernel space allocated for it and will be initialized to load at that location when referenced.

-A
: Causes **kern_loader** to shut down; all existing kernel servers are unloaded and deallocated, and the running **kern_loader** task exits.

-d *server_name ...*
: Causes **kern_loader** to deallocate the specified kernel server or servers; all physical and virtual resources associated with the kernel server are freed.

-l *server_name ...*
: Causes **kern_loader** to load the specified kernel server or servers into the kernel. If you don't use this option, loading is normally done either when the kernel server is allocated (if START is specified in the load commands), or when it receives its first message.

-L
: Causes **kl_util** not to terminate at the end of its operation, so that further **kern_loader** activity can be monitored. As long as **kl_util** is running, anything logged by **kern_loader** is displayed.

-r
: Causes **kern_loader** to deallocate all its servers and set itself up from scratch by rereading its configuration file. This is similar to specifying the **-A** option and then restarting **kern_loader**, except that **kern_loader** never actually exits.

-s [*server_name ...*]
: Causes **kern_loader** to return information about the status of registered kernel servers. If a server name isn't specified, a list of all known servers is displayed. If a server name is specified, detailed information about that server is displayed.

  The following example shows the status of the MIDI driver:

```
# /usr/etc/kl_util -s mididriver
SERVER: mididriver
RELOCATABLE: /usr/lib/kern_loader/Midi/mididriver_reloc
STATUS: Allocated at address 0x10f70000 for 0xa000 bytes
PORTS: mididriver(advertised)
#
```

-u *server_name ...*

Causes **kern_loader** to unload the specified kernel server or servers. (Loaded kernel servers remain in the kernel until they're explicitly unloaded.) Unloading the server causes any wired pages to be unwired; thus, this can be used as a mechanism to free up resources in the system when the server is no longer needed.

Kernel-Server Log Command (kl_log)

You can use the kernel-server log command, **kl_log**, to see log messages from a loadable kernel server. You can also write your own program that calls the **kern_loader_log_level()** and **kern_loader_get_log()** functions to get log messages. The **kern_loader_log_level()** and **kern_loader_get_log()** functions are discussed in Chapter 3, "Using Loadable Kernel Servers."

You must be superuser to call **kl_log**. It has the following syntax:

/usr/etc/kl_log [**-l** *log_level*] *server_name*

where:

server_name Specifies the loadable kernel server for which you're getting or setting log information. This server must already be loaded.

-l *log_level* Specifies the priority of messages that should be kept. By default, the log level is zero, and no log messages are printed. By setting *log_level* to a positive value, you ensure that log messages from the server will be printed to **stdout** if they have a priority equal to or greater than *log_level.*

You might use **kl_log** as follows:

```
slave# kl_log -l 1 mydriver&
.
.
.
slave# kl_log -l 0 mydriver
slave# jobs
[1]   + Running                 kl_log -l 1 mydriver
slave# kill %1
slave#
```

Before you stop collecting messages from a kernel server, you should shut off logging by setting its log level to zero. If you don't set the log level to zero, log messages will accumulate even though no process is collecting the messages.

B *The ROM Monitor and NMI Mini-Monitor*

Most of the commands in the ROM monitor and NMI mini-monitor are discussed in the *NeXTSTEP Network and System Administration* manual; some other useful commands are covered in Chapter 9 of this book, "Building, Loading, and Debugging Loadable Kernel Servers." However, a few commands are of interest to only a few hardware or device-driver developers. These rarely used commands are described in this appendix.

ROM Monitor Commands

Most ROM monitor commands display the current value of a register or parameter, and prompt for a new value with a question mark. To enter a new value, type the value and press the Return key. To leave the current value unchanged and skip to the next one (if any), just press the Return key. Type a period to exit the command and leave the current value unchanged.

Numeric values are usually in hexadecimal (base 16) notation if they represent a memory address or data value. The **R** (radix) command can be used to change the input base/radix.

The values of register bit fields are displayed symbolically between angle brackets **< >** in addition to the numeric value (for example, "2700<trace=0,s,ipl=7>"). For more information on 68040 registers, see Motorola's *MC68040 User's Manual.*

Open Address Register

The command

a [*n*]

accesses the contents of address registers a0 through a7 of the 680x0 processor.

Example:

```
NeXT> a
a0: 00001234? 5678          Change a0 from 0x1234 to 0x5678.
a1: 00000000?               Don't change a1.
a2: 00000001? .             Type a period to exit.

NeXT> a0
a0: 00001234? 5678          Change a0 from 0x1234 to 0x5678.
NeXT>
```

Open Data Register

The command

d [*n*]

accesses the contents of data registers d0 through d7 of the 680x0 processor.

Example:

```
NeXT> d
d0: 00001234? 5678          Change d0 from 0x1234 to 0x5678.
d1: 00000000?               Don't change d1.
d2: 00000001? .             Type a period to exit.

NeXT> d0
d0: 00001234? 5678          Change d0 from 0x1234 to 0x5678.
NeXT>
```

Open Processor Register

The command

r [*regname*]

accesses the contents of the 680x0 processor registers. Possible values for *regname* are:

pc	Program counter
sr	Status register
usp	User stack pointer
isp	Interrupt stack pointer
msp	Master stack pointer
vbr	Vector base register
sfc	Source function code
dfc	Destination function code
cacr	Cache control register
caar	Cache address register (68030 only)

If you don't specify *regname*, all the processor registers are opened in the order listed above.

Open System Register

The command

s [*systemreg*]

accesses the contents of the system registers. Possible values for *systemreg* are:

intrstat	Interrupt status register
intrmask	Interrupt mask register
scr1	System control register #1
scr2	System control register #2

If you don't specify *systemreg*, all of the system registers are opened in the order they're listed above.

Examine Memory Locations

The command

e [*lwb*] [*addrlist*] [*format*]

lets you examine particular locations in memory.

[*lwb*]	Specify **l**, **w**, or **b** to select long, word, or byte length. The default is long.
[*addrlist*]	This argument specifies the starting address or list of addresses to cyclically examine. If you don't specify a value for *addrlist*, the command uses the most recent value of *addrlist*.
[*format*]	This argument controls how the value is displayed. It can be any of the standard format types supported by the C language **printf()** library routine. The default, **%x**, displays a number in hexadecimal.

To have access to hardware addresses, you must get into the ROM monitor just after the system is powered on, before it has booted. Otherwise, you'll read or write to virtual addresses, not physical addresses. To enter the ROM monitor, do the following:

1. If your computer is on, press the Power key to turn it off.

Locate the ~ key on your keyboard. (You'll need to press it along with a Command key soon after you turn the power back on.)

2. Turn the computer back on by pressing the Power key.

3. *Within three seconds* of the appearance of the message "Loading from disk" or "Loading from network," hold down the Command key and press the ~ key.

You should now be in the ROM monitor. You can use the **e** command to look at addresses on the CPU board.

Example 1:

`NeXT> e 4000000`	Examine the long at memory location 0x4000000.
`4000000: 0? 12345678`	See that its value is 0x0; deposit the value 0x12345678.
`4000004: 0? .`	Type a period to exit.

Example 2:

`NeXT> e 4000000 4000000`	Repeatedly examine memory location 0x4000000.
`4000000: 0? 12345678`	See that its value is 0x0; deposit the value 0x12345678.
`4000000: 12345678? .`	See that its value is 0x12345678; type a period to exit.

Open Function Code

The command

S [*fcode*]

lets you inspect or modify the 680x0 function code (address space) used with the **e** (examine) command. The default code is 5 (supervisor data space).

Possible values for *fcode* are:

0	Undefined, reserved
1	User data space (UD)
2	User program space (UP)
3	Undefined, reserved
4	Undefined, reserved
5	Supervisor data space (SD)
6	Supervisor program space (SP)
7	CPU space

Example:

```
NeXT> s
Function code 5 (SD)          Current function code is 5.
NeXT> s 6
Function code 6 (SP)          Change function code to 6.
```

Set Input Radix

The command

R [*radix*]

lets you set the input radix. The default value for the input radix is 16. Any numbers you enter are interpreted in the base of the input radix (for example, input radix 16 means numbers are interpreted as base 16). If you don't specify *radix*, the current default input radix is displayed.

NMI Mini-Monitor Commands

The following NMI mini-monitor commands aren't currently useful for developers outside of NeXT.

Set or Examine Any Kernel Flag

The command

flag=[*value*]

lets you examine or modify internal system flags (most internal system flags are of interest only to system developers at NeXT).

Example:

```
nmi> debug=
kernel flags debug = 0x0
nmi> debug=2
kernel flag = 0x2
```

The **debug** flag has the value 0.

Change the value of the **debug** flag to 2.

Set the IP Address

The command

kd_ipaddr=*value*

lets you set the IP address used by the computer for kernel debugging. It's useful only to system developers at NeXT.

C *Summary of Kernel Support Functions*

This appendix summarizes the kernel support functions (and some macros that behave like functions) that loadable kernel servers can call. Within the general categories of "General Functions" and "Network Functions," function declarations are further subgrouped to help you identify their interrelationships.

Chapter 10, "Kernel Support Functions," contains full descriptions of all the functions listed here. In addition, loadable kernel servers can use many Mach kernel functions, which are described in a section of Chapter 4, "Mach Functions." The Mach kernel functions are summarized in the manual, *NeXTSTEP Programmer Interface*.

General Functions

This section contains a summary of the general purpose kernel support functions. Most of the functions and macros in this section are declared through either the **kernserv/kern_server_types.h** or **kernserv/prototypes.h** header file.

Time Functions

Busy-wait for a certain amount of time:

void	**DELAY**(unsigned int *usecs*)

Get or set the current time:

ns_time_t	**clock_value**(clock_types_t *which_clock*)
void	**set_clock**(clock_types_t *which_clock*, ns_time_t *ns*)

Get information about a clock:

chrono_attributes_t	**clock_attributes**(clock_types_t *which_clock*)

Convert between **ns_time_t** and **timeval** data formats:

void	**ns_time_to_timeval**(ns_time_t *ns*, struct timeval **tv*)
ns_time_t	**timeval_to_ns_time**(struct timeval **tv*)

Schedule or unschedule a function to be called later:

void	**ns_abstimeout**(func *function*, vm_address_t *arg*, ns_time_t *deadline*, int *priority*)
void	**ns_timeout**(func *function*, vm_address_t *arg*, ns_time_t *time*, int *priority*)
boolean_t	**ns_untimeout**(func *function*, vm_address_t *arg*)

Memory Functions

Make addresses pageable or memory-resident:

kern_return_t	**kern_serv_unwire_range**(kern_server_t **ksp*, vm_address_t *address*, vm_size_t *size*)
kern_return_t	**kern_serv_wire_range**(kern_server_t **ksp*, vm_address_t *address*, vm_size_t *size*)

Copy or initialize data:

void	**bcopy**(void **from*, void **to*, int *length*)
void	**bytecopy**(void **from*, void **to*, int *length*)
void	**bzero**(void **address*, int *length*)

Allocate or free memory:

void *	**kalloc**(int *size*)
void	**kfree**(void **address*, int *size*)
void *	**kget**(int *size*)

Critical Section and Synchronization Functions

Use read and write locks:

lock_t	**lock_alloc**(void)
void	**lock_free**(lock_t *lock*)
void	**lock_done**(lock_t *lock*)
void	**lock_init**(lock_t *lock*, boolean_t *can_sleep*)
void	**lock_read**(lock_t *lock*)
void	**lock_write**(lock_t *lock*)

Use simple, nonsleeping locks:

void	**simple_lock**(simple_lock_t *lock*)
simple_lock_t	**simple_lock_alloc**(void)
void	**simple_lock_free**(simple_lock_t *lock*)
void	**simple_lock_init**(simple_lock_t *lock*)
void	**simple_unlock**(simple_lock_t *lock*)

Cause a thread to sleep or wake up:

void	**assert_wait**(int *event*, boolean_t *interruptible*)
void	**clear_wait**(thread_t *thread*, int *result*, boolean_t *interrupt_only*)
void	**thread_block**(void)
void	**thread_set_timeout**(int *ticks*)
void	**thread_sleep**(int *event*, simple_lock_t *lock*, boolean_t *interruptible*)
void	**thread_wakeup**(int *event*)

General Task and Thread Functions

Get information about this thread or task:

task_t	**current_task**(void)
int	**thread_wait_result**(void)

Create or kill a thread:

thread_t	**kernel_thread**(task_t *task*, void (**start*)(void))
void	**thread_halt_self**(void)

Port and Message Functions

Request notification messages, such as port death notification:

kern_return_t **kern_serv_notify**(kern_server_t **ksp*, port_t *reply_port*, port_t *request_port*)

Get the kernel's task port:

port_t **kern_serv_kernel_task_port**(void)

Get or set information about this server's ports:

port_t **kern_serv_bootstrap_port**(kern_server_t **ksp*)
port_t **kern_serv_local_port**(kern_server_t **ksp*)
port_t **kern_serv_notify_port**(kern_server_t **ksp*)
void **kern_serv_port_gone**(kern_server_t **ksp*, port_name_t *port*)
kern_return_t **kern_serv_port_proc**(kern_server_t **ksp*, port_all_t *port*, port_map_proc_t *function*, int *arg*)
kern_return_t **kern_serv_port_serv**(kern_server_t **ksp*, port_all_t *port*, port_map_proc_t *function*, int *arg*)
port_set_name_t **kern_serv_port_set**(kern_server_t **ksp*)

Hardware Interface Functions

Set up or remove an interrupt handler:

int **install_polled_intr**(int *which*, int (**my_intr*)(void))
int **uninstall_polled_intr**(int *which*, int (**my_intr*)(void))

Get or test a virtual address that corresponds to a hardware address:

caddr_t **map_addr**(caddr_t *address*, int *size*)
int **probe_rb**(void **address*)

Change or determine the processor level:

int **curipl**(void)
int **spl0**(void), **spl1**(void), **spl2**(void), **spl3**(void), **spl4**(void), **spl5**(void), **spl6**(void), **spl7**(void)
void **splx**(int *priority*)

Logging and Debugging Functions

Kill the loadable kernel server:

void	**ASSERT**(int *expression*)
kern_return_t	**kern_serv_panic**(port_t *bootstrap_port*, panic_msg_t *message*)
void	**panic**(char **string*)

Log a message:

void	**kern_serv_log**(kern_server_t **ksp*, int *log_level*, char **format*, *arg1*, ..., *arg5*)
int	**log**(int *level*, char **format*, *arg*, ...)
int	**printf**(char **format*, *arg*, ...)

UNIX Support Functions

In a UNIX-style server, determine whether the user has root privileges:

int	**suser**(void)

In a UNIX-style server, wait for I/O completion on a buffer:

void	**biodone**(struct buf **bp*)
void	**biowait**(struct buf **bp*)

In a UNIX-style server, copy data between user and kernel address space:

int	**copyin**(void **from*, void **to*, int *length*)
int	**copyout**(void **from*, void **to*, int *length*)

In a UNIX-style server, implement the **select()** system call:

int	**selthreadcache**(void ***waiterPtr*)
void	**selthreadclear**(void ***waiterPtr*)
int	**selwakeup**(void **waiter*, int *collided*)

Miscellaneous Functions

Modify or inspect a string:

int	**sprintf**(char **string*, char **format*, *arg*, ...)
char *	**strcat**(char **string1*, char **string2*)
int	**strcmp**(char **string1*, char **string2*)
char *	**strcpy**(char **to*, char **from*)
int	**strlen**(char **string*)

Call a function from the main thread:

kern_return_t	**kern_serv_callout**(kern_server_t **ksp*, void (**func*)(void *), void **arg*)

Network Functions

This section contains a summary of the network-specific kernel support functions, which are described in detail in Chapter 10. A general discussion of networking drivers and protocols is in Chapter 8, "Network Modules."

Netif Functions

To use these functions, you need to include the header file **net/netif.h**.

Initialize and install a new netif:

netif_t	**if_attach**(if_init_func_t *init_func*, if_input_func_t *input_func*, if_output_func_t *output_func*, if_getbuf_func_t *getbuf_func*, if_control_func_t *control_func*, const char **name*, unsigned int *unit*, const char **type*, unsigned int *mtu*, unsigned int *flags*, netif_class_t *class*, void **private*)
void	**if_registervirtual**(if_attach_func_t *attach_func*, void **private*)

Remove a netif:

void	**if_detach**(netif_t *netif*)

Get or set data for a netif:

unsigned int	**if_collisions**(netif_t *netif*)
void	**if_collisions_set**(netif_t *netif*, unsigned int *collisions*)
unsigned int	**if_flags**(netif_t *netif*)
void	**if_flags_set**(netif_t *netif*, unsigned int *flags*)
unsigned int	**if_ierrors**(netif_t *netif*)
void	**if_ierrors_set**(netif_t *netif*, unsigned int *ierrors*)
unsigned int	**if_oerrors**(netif_t *netif*)
void	**if_oerrors_set**(netif_t *netif*, unsigned int *oerrors*)
unsigned int	**if_ipackets**(netif_t *netif*)
void	**if_ipackets_set**(netif_t *netif*, unsigned int *ipackets*)
unsigned int	**if_opackets**(netif_t *netif*)
void	**if_opackets_set**(netif_t *netif*, unsigned int *opackets*)
unsigned int	**if_mtu**(netif_t *netif*)
const char *	**if_name**(netif_t *netif*)
void *	**if_private**(netif_t *netif*)
const char *	**if_type**(netif_t *netif*)
unsigned int	**if_unit**(netif_t *netif*)

Call a function implemented by a network module:

int	**if_control**(netif_t *netif*, const char **command*, void **data*)
netbuf_t	**if_getbuf**(netif_t *netif*)
int	**if_init**(netif_t *netif*)
int	**if_ioctl**(netif_t *netif*, unsigned int *command*, void **data*)
int	**if_output**(netif_t *netif*, netbuf_t *packet*, void **address*)

Get information about netifs:

netif_class_t	**if_class**(netif_t *netif*)
netif_t	**iflist_first**(void)
netif_t	**iflist_next**(netif_t *netif*)

Dispatch a packet to a protocol handler:

int	**if_handle_input**(netif_t *netif*, netbuf_t *packet*, void **extra*)

Netbuf Functions

You should include the header file **net/netbuf.h** when you use these functions.

Allocate or free a netbuf or its wrapper:

netbuf_t	**nb_alloc**(unsigned int *size*)
netbuf_t	**nb_alloc_wrapper**(void **data*, unsigned int *size*, void *freefunc*(void *), void **freefunc_arg*)
void	**nb_free**(netbuf_t *nb*)
void	**nb_free_wrapper**(netbuf_t *nb*)

Change the size of a netbuf:

int	**nb_grow_bot**(netbuf_t *nb*, unsigned int *size*)
int	**nb_shrink_bot**(netbuf_t *nb*, unsigned int *size*)
int	**nb_grow_top**(netbuf_t *nb*, unsigned int *size*)
int	**nb_shrink_top**(netbuf_t *nb*, unsigned int *size*)

Access the data in a netbuf:

char *	**nb_map**(netbuf_t *nb*)
int	**nb_read**(netbuf_t *nb*, unsigned int *offset*, unsigned int *size*, void **target*)
int	**nb_write**(netbuf_t *nb*, unsigned int *offset*, unsigned int *size*, void **source*)
unsigned int	**nb_size**(netbuf_t *nb*)

Miscellaneous Functions

For the host-network conversion functions, you need to include the header file **netinet/in.h**. For **inet_queue()**, you should include both **net/netif.h** and **net/netbuf.h**.

Convert values between host and network byte order:

u_long	**htonl**(u_long *hostlong*)
u_short	**htons**(u_short *hostshort*)
u_long	**ntohl**(u_long *netlong*)
u_short	**ntohs**(u_short *netshort*)

Give an IP input packet to the kernel for processing:

void	**inet_queue**(netif_t *netif*, netbuf_t *netbuf*)

Glossary

Application Kit
The Objective C classes and C functions available for implementing the NeXT window-based user interface in an application. The Application Kit provides a basic program structure for applications that draw on the screen and respond to events.

board address space
256 megabytes per slot of physical address space, from 0x*s*0000000 to 0x*s*fffffff, where *s* is the slot number.

bootstrap port
A port to which a new task can send a message that will return any other system service ports that the task needs.

condition variable
A type of variable provided by the C-thread functions to help synchronize the threads in a task.

console
A special window that displays system log messages, as well as output written to the standard error and standard output streams by applications launched from the Workspace Manager.

demand paging
An operating system facility that causes pages of data to be brought from disk into physical memory only as they are needed.

device driver
A thread running in the kernel's address space that supports a specific device. Some NeXT drivers and all non-NeXT drivers are implemented as loadable kernel servers.

DMA
Direct memory access; a means of transferring data between host memory and a peripheral device without involving the host processor.

DSP
Digital signal processor, a device that modifies digital signals; on the NeXT computer, the Motorola DSP56001 microprocessor.

Ethernet
A high-speed local area network technology. Ethernet is considered the industry standard for networking UNIX-based machines because of its reliability and capacity to rapidly transfer large amount of information. Ethernet connectors are built into NeXT computers.

exception
A synchronous interruption to the normal flow of program control caused by the program itself.

exception port
In Mach, a port on which a task or thread receives messages when exceptions occur.

host
The computer that's running (is host to) a particular program. The term is usually used to refer to a computer on a network.

host processor
The microprocessor on which an application program resides. When an application is running, the host processor may call other, peripheral microprocessors, such as the DSP56001, to perform specialized operations.

inheritance attribute
In Mach, a value indicating the degree to which a parent process and its child process share the parent process's address space. A memory page can be inherited copy-on-write, shared, or not at all.

in-line data
Data that's included directly in a Mach message, as opposed to referred to by a pointer. See also *out-of-line data.*

instance variable
In a loadable kernel server, the ordinary C variable of type **kern_server_t** that the kernel-server loader uses to keep track of the loadable kernel server.

I/O
Input/output; the sending and retrieving of information into the memory of a program, usually to and from a file or a peripheral device through an I/O port.

IPC
Interprocess communication; the transfer of information between processes. In Mach, IPC is performed through the use of messages.

kernel port
A port used to represent a task or thread in Mach function calls. Also known as a task port or thread port.

loadable kernel server
In the NeXT Mach operating system, a software module that's loaded into the kernel after the system has been booted. See also *device driver* or *network module*.

loadable object file
For loadable kernel servers, an object file that has been linked against the kernel. Loadable object files are required for debugging loadable kernel servers with GDB. See also *relocatable object file*.

Mach
The multitasking operating system used by all NeXT computers. Mach is completely compatible with UNIX 4.3BSD but adds faster interprocess communication, a larger virtual memory space, memory-mapped files, and multiple threads of execution within a single address space.

Mach factor
A measurement of how busy the system is. Unlike the UNIX load average, higher Mach factors mean that the system is less busy.

Mach server
A task that provides services to clients, using a MiG-generated RPC interface.

makefile
A specification file used by the program **make** to build an executable version of an application. A makefile details the files, dependencies, and rules by which the application is built.

master computer
When debugging a loadable kernel server with GDB, the master computer is the one on which GDB is running. The master computer uses GDB to watch the slave computer execute the server being debugged.

memory-mapped files
A Mach facility that maps virtual memory onto a physical file. Thereafter, any reference to that part of virtual memory causes the corresponding page of the physical file to be brought into memory.

message
In Mach, a message consists of a header and a variable-length body; operating system services are invoked by passing a message from a thread to the port representing the task that provides the desired service.

MIDI
Musical Instrument Digital Interface; the industry standard used by modern keyboard synthesizers for transmitting and storing musical performance information.

MiG
Mach's message interface generator. MiG provides a procedure call interface to Mach's system of interprocess messaging.

multitasking
Describes an operating system that allows the concurrent execution of multiple programs. Mach, the operating system of all NeXT computers, is multitasking.

mutex variable
Mutual exclusion variable; a type of variable provided by the C-thread functions to help protect critical regions in a multiple-thread task.

NBIC (NeXTbus Interface Chip)
The chip that lets the CPU board communicate with the NeXTbus. NeXTbus boards can also use this chip to implement their interface to the NeXTbus.

netbuf
Network buffer; a data structure that network modules use for network packet buffers.

netif
Network interface; a data structure used to register and refer to network modules.

network
A group of hosts that can directly communicate with each other.

network module
A loadable kernel server that performs a network-related function. See also *protocol handler*, *packet sniffer*, and *device driver*.

network port
In Mach, a port by which local objects communicate with remote objects. A message sent to a network port is received by the local network server, processed, and then sent across the network to a remote network server.

network port identifier
A code by which a network server determines the identity of the recipient local task.

network server
A local operating system representative for tasks on a remote computer. Messages intended for a remote task are processed and redirected by a local network server.

NeXTSTEP
NeXT's application development and user environment, consisting of the Workspace Manager, various applications such as Project Builder and Interface Builder, various software kits such as the Application Kit and the Database Kit, and the Window Server.

NFS
Network File System. An NFS file server allows users on the network to share files as if they were on their own local disk.

NMI
Non-maskable interrupt; an interrupt produced by a particular keyboard sequence.

nonsimple message
In Mach, a message that contains either a reference to a port or a pointer to data.

notify port
In Mach, a port on which a task receives messages from the kernel advising it of changes in port access rights and of the status of messages it has sent.

out-of-line data
Data that's passed by reference in a Mach message, as opposed to being included in the message. See also *in-line data.*

packet
An individual piece of information sent on a TCP/IP network.

packet sniffer
A network module that examines input packets for diagnostic purposes. See also *protocol handler.*

paging object
In Mach, a secondary storage object that's mapped into a task's virtual memory. See also *demand paging*.

physical address
An address to which a hardware device, such as a memory chip or a NeXTbus board, can directly respond. Programs, including the Mach kernel and loadable kernel servers, use virtual addresses that are translated to physical addresses by mapping hardware controlled by the Mach kernel. See also *board address space*, *slot address space*, and *virtual address*.

policy
In Mach, a thread's scheduling policy determines how the thread's priority is set and under what circumstances the thread runs. See also *priority*.

port
In Mach, a protected communication channel by which messages are sent to, and received from, operating system objects.

port access rights
In Mach, the ability to send to or receive from a port.

port set
In Mach, a set of zero or more ports. A thread can receive messages sent to any of the ports contained in a port set by specifying the port set as a parameter to **msg_receive**().

priority
In Mach scheduling, a number between 0 and 31 that indicates how likely a thread is to run. The higher the thread's priority, the more likely the thread is to run. See also *policy*.

process
A program that is at some stage of execution. In Mach, a task containing a single thread of execution is equivalent to a process.

process identifier,* or *process ID
In UNIX, a number that uniquely identifies a process.

programmed I/O
Byte-by-byte or word-by-word data transfer to a device. Also known as "direct I/O." See also *DMA*.

protocol handler
A network module that extracts data from input packets (giving the data to interested programs) and inserts data into output packets (giving the output packet to the appropriate network device driver).

quantum
The fixed amount of time a thread can run before being preempted.

RAM
Random-access memory; memory that a microprocessor can either read or write to.

real time
A concept of time when using a computer. If the user defines or initiates an event and the event occurs instantaneously, the computer is said to be operating in real time.

receive rights
In Mach, the ability to receive messages on a port. Only one task at a time can have receive rights for any one port. See also *send rights*.

relocatable object file
For loadable kernel servers, the object file that the kernel-server loader uses to load the server into the kernel. See also *loadable object file*.

reply port
A port associated with a thread that's used in Mach remote procedure calls.

ROM
Read-only memory; memory that a microprocessor can read but not write to.

RPC
Remote procedure call; in Mach, RPCs are implemented using MiG-generated messages.

SCSI
Small Computer Systems Interface. A standard connector and communications protocol used for connecting devices such as disk drives to computers.

send rights
In Mach, the ability to send messages to a port. Many tasks can have send rights for the same port. See also *receive rights*.

server
See *loadable kernel server*, *Mach server*, or *network server*.

simple message
In Mach, a message that contains neither references to ports nor pointers to data.

slave computer
When debugging a loadable kernel server with the GNU source-level debugger (GDB), the slave computer is the one on which the driver being debugged is running. The execution of the slave's kernel is watched by the master computer.

slot address space
16 megabytes per slot of physical address space, from 0x*fs*000000 to 0x*fs*ffffff, where *s* is the slot number.

Sound Kit
The Objective C classes and C functions available for creating sound effects, doing speech analysis, and performing other sound manipulation.

task
In Mach, a paged virtual address space along with protected access to ports, virtual memory, and system processor(s). A task itself performs no computation; rather, it's a framework for running threads. See also *thread*.

task port
In Mach, a port by which all threads within a task may be addressed. Also known as the task's kernel port.

TCP/IP
Transmission Control Protocol/Internet Protocol. The protocols used to deliver messages between computers over the network. TCP/IP support is included in NeXT computers.

thread
In Mach, the basic unit of program execution. A thread consists of a program counter, a set of registers, and a stack pointer. See also *task*.

thread port
In Mach, a port that represents a single thread within a task. Also known as the thread's kernel port.

thread-safe
Code that can be used safely by several threads simultaneously.

virtual address
An address that is usable by software. Each task on a NeXT computer has its own range of virtual addresses, which begins at address zero. The Mach operating system makes the CPU hardware map these addresses onto physical memory only when necessary, using disk memory at other times. See also *physical address*.

Window Server
A process that dispatches user events to applications and renders PostScript code on behalf of applications.

wired down
Always resident in memory. A region of virtual memory that's wired down can't be swapped or paged out. Memory must be wired down for interrupt handlers and the hardware to use it safely.

Suggested Reading

Some information you may need or find useful isn't covered in detail in this manual. This section indicates where you can get additional information, whether in printed form or on-line.

Programming Tools and the C Language

NeXTSTEP Development Tools and Techniques: Release 3. NeXT Publications. Addison-Wesley, 1992.

The above manual has information about the compiler and debugger, among other tools.

American National Standard X3.159-1989, Programming Language C.

This document is the formal and official definition of the C language, its preprocessor, and run-time library. It's available directly from the American National Standards Institute. To order by telephone, call (212)642-4900.

ANSI C: A Lexical Guide. Mark Williams Company. Prentice-Hall, 1988.

This ANSI C handbook is organized in a dictionary-style format that provides an explanation of every C program construct; it also includes a C tutorial.

The C Programming Language. Second edition. Brian W. Kernighan and Dennis M. Ritchie. Prentice-Hall, 1988.

This edition is based on the ANSI standard for C.

C: A Reference Manual. Second edition. Samuel P. Harbison and Guy L. Steele. Prentice-Hall, 1987.

This edition includes a summary of the Draft Proposed ANSI C standard and a complete description of the run-time library.

Portability and the C Language. Rex Jaeschke. Hayden Books, 1989.

This reference work covers language, preprocessor, and standard run-time library issues, as well as discussing the impact of the ANSI C standard.

Mach Operating System

Operating System Concepts. Third edition. Abraham Silberschatz, James Peterson, and Peter Galvin. Addison-Wesley, 1991.

Modern Operating Systems. Andrew Tanenbaum. Prentice-Hall, 1992.

Each of the two books listed above has a chapter about the Mach operating system.

The Design of the Mach Operating System. Nawaf Bitar, Alan Langerman, and Eric Shienbrood. Prentice-Hall, expected 1993.

Carnegie Mellon University makes papers, conference and course announcements, and other information about Mach available by anonymous FTP. (See the UNIX manual page for **ftp** for information on how to use FTP.) Be aware that NeXT is based on Mach 2.0 and hasn't incorporated all the features of Mach 2.5. NeXT has also added some features such as loadable kernel servers and the Bootstrap Server to the Mach operating system.

UNIX 4.3BSD Operating System

The UNIX Programming Environment. Brian W. Kernighan and Rob Pike. Prentice-Hall, 1984.

The above book is one of the many that describe how to use the UNIX operating system.

The Design of the UNIX Operating System. Maurice J. Bach. Prentice-Hall, 1986.

The Design and Implementation of the 4.3BSD UNIX Operating System. Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. Addison-Wesley, 1989.

Writing a UNIX Device Driver. Janet I. Egan and Thomas J. Teixeira. John Wiley & Sons, Inc., 1988.

The above book has good information for any driver writer. It also has specific information on UNIX drivers.

Standard UNIX® manual pages contain information about UNIX commands and system calls. These are stored on-line, and are accessible in Digital Librarian.

Networking

Computer Networks: Protocols, Standards, and Interfaces. Uyles Black. Prentice-Hall, 1987.

Internetworking with TCP/IP. Volume I: Principles, Protocols, and Architectures. Douglas Comer. Prentice-Hall, 1991.

Internetworking with TCP/IP. Volume II: Design, Implementation, and Internals. Douglas Comer and David L. Stevens. Prentice-Hall, 1991.

Microprocessors

MC68040 User's Manual. Motorola, Inc., 1989.

MC68030 Enhanced 32-Bit Microprocessor User's Manual. Motorola, Inc. Prentice-Hall, 1987.

NeXTSTEP System Administration

NeXTSTEP Network and System Administration: Release 3. NeXT Publications. Addison-Wesley, 1992.

NeXTSTEP General Reference: Release 3. NeXT Publications. Addison-Wesley, 1992.

NeXTSTEP Programming Interface Summary: Release 3. NeXT Publications. Addison-Wesley, 1992.

The two books above present the programming interface for NeXTSTEP applications. See the back cover of this manual for more titles in the *NeXTSTEP Developer's Library.*

Index

a ROM monitor command B-2
access rights *See* port access rights
add-file GDB command 9-11
address space
    copying bytes from kernel to user 10-11
    copying bytes from user to kernel 10-10
    NeXTbus 7-2, 7-3
    physical 1-13, 7-2, 7-3
    regions of 1-14
    virtual 1-13, 1-14
addresses
    checking existence of 10-37
    converting from physical to virtual 7-4, 10-31
ADVERTISE load command 9-3, A-5
Application Kit 1-25
    thread safety of 1-27
applications, multithreaded 1-6, 1-26, 4-2
array MiG keyword 2-20
ASSERT() 6-5, 9-2, 10-2
assert_wait() 10-3

backtrace GDB command 9-6, 9-11
bcopy() 10-4
biodone() 10-4
biowait() 10-5
block device entry points 6-16
board address space, NeXTbus 7-2
boot ROM monitor command 9-8
bootstrap ports 1-7
    creating new 4-124
Bootstrap Server 2-9
Bootstrap Server functions 4-115–4-125
bootstrap_check_in() 4-115
bootstrap_create_service() 4-116
bootstrap_info() 4-117
bootstrap_look_up() 2-10, 4-119
bootstrap_look_up_array() 4-120
bootstrap_register() 4-121
bootstrap_status() 4-123
bootstrap_subset() 4-124
buffers
    copying kernel data into 10-4, 10-6
    waiting until a function calls **biodone**() on 10-5
    waking up functions doing a **biowait**() on 10-4
BusProbe 7-4, 9-6
byte ordering
    converting between host and network 10-54
    NeXTbus 7-6
bytecopy() 10-6
bzero() 10-7

C-thread functions 1-26, 4-2–4-21

C threads
associating data with 4-9
associating strings with 4-14
counting the number in a task 4-9
creating 1-26, 4-10, 4-12
getting the C thread identifier of 4-17
getting the errno value of 4-11
getting the Mach thread identifier of 4-18
interrupting 4-8
joining 1-26, 4-13
limiting the number in a task 4-13
programming example 1-30
protecting critical regions 1-26
setting priorities of 4-15
setting the errno value of 4-18
suspending 4-8
synchronizing 1-26
terminating 4-11
waking up 4-3, 4-7
yielding the processor 4-19
CALL load command A-5
CALL unload command A-7
callback functions in network modules 8-4, 10-63
calloc() 1-16, 1-17
catch_exception_raise() 1-43, 1-44, 4-22
character device entry points 6-13
clear_wait() 10-7
clients, of Mach servers 2-10
compiling 2-11
clock_attributes() 10-8
clock_value() 10-9
condition variables 1-28
associating strings with 4-6
clearing 4-4
creating 4-2
destroying 4-4
initializing 4-5
use of 1-28
condition_alloc() 4-2
condition_broadcast() 4-3
condition_clear() 4-4
condition_free() 4-4
condition_init() 4-5
condition_name() 4-6
condition_set_name() 4-6
condition_signal() 4-7
condition_wait() 1-29, 4-8
console user name 9-6
console window 6-5
opening 9-6
cont GDB command 9-11
continue NMI mini-monitor command 9-5
control functions in network modules 8-4, 8-11, 10-57
Control register, NBIC 7-5, 7-9
copyin() 6-11, 10-10
copy-on-write memory 1-14, 1-20
copyout() 6-11, 10-11
CPU, how to reset 9-4
cthread_abort() 4-8
cthread_count() 4-9
cthread_data() 4-9
cthread_detach() 4-10
cthread_errno() 1-26, 4-11
cthread_exit() 4-11
cthread_fork() 1-27, 4-12
cthread_join() 4-13
cthread_limit() 4-13
cthread_max_priority() 1-24, 4-15
cthread_name() 4-14
cthread_priority() 1-24, 4-15
cthread_self() 4-17
cthread_set_data() 4-9
cthread_set_errno_self() 4-18
cthread_set_limit() 4-13
cthread_set_name() 4-14
cthread_thread() 4-2, 4-18
cthread_yield() 4-19
curipl() 10-11
current_task() 10-12

d ROM monitor command B-2
deadlock, avoiding 1-29
dealloc MiG keyword 2-17, 2-23
DEBUG C preprocessor macro 6-5, 9-2
debugging loadable kernel servers 6-4, 9-4

DELAY() 10-12
delete GDB command 9-12
demand paging 1-13
detach GDB command 9-12
DETACH load command A-6
device drivers
See also loadable kernel servers
network *See* network modules
NeXTbus 7-1
supplied by NeXT 5-3
dir GDB command 9-11
direct I/O 7-5
direct memory access (DMA) 7-5
Distributed Objects 4-115, 4-126
for interprocess communication 2-1
thread safety of 1-26
DMA 7-5
DPS client routines, thread safety of 1-27
drivers *See* loadable kernel servers *or* device drivers

e ROM monitor command B-4
entry points
block device 6-16
character device 6-13
errno variable 1-26
error MiG keyword 2-22, 2-26
error strings
displaying or getting 4-35, 4-133
logging from loadable kernel servers 10-17
Ethernet interfaces 8-12
exc_server() 1-43, 1-44, 4-22
exception handling 1-32
See also exceptions
in debuggers 1-34, 1-39
displaying or getting exception strings 4-36
in error handlers 1-34, 1-38
implementing 1-38, 1-39
in Mach 1-35
model for Mach 1-34
programming example 1-44
thread safety of 1-26
in UNIX 1-33
exception handling functions 1-43, 4-22–4-27
exception ports 1-7, 1-36, 1-39
dispatching messages received on 4-22
exception_raise() 1-43, 1-44, 4-26
exceptions 1-7
See also exception handling
causing 4-26
classes of 1-42
user-defined 1-37

files, memory-mapped 1-16
fixed-priority policy 1-25
free() 1-16, 1-17
function MiG keyword 2-22
functions
available to loadable kernel servers 6-3
Bootstrap Server 4-115–4-125
C-thread 1-26, 4-2–4-21
exception handling 1-43, 4-22–4-27
implemented in network modules 8-4
kernel-server loader 3-3, 4-129–4-154, A-2
kernel support 10-1–10-69, C-1
Mach 4-1–4-154
Mach kernel 4-22–4-114, 6-3
netbuf 10-65–10-69, C-8
netif 10-54–10-64, C-6
network 10-54–10-69, C-6
Network Name Server 4-126–4-128
running in the loadable kernel server's main thread 10-15
scheduling for a future time from loadable kernel servers 10-32
unscheduling a call arranged by **ns_timeout()** or **ns_abstimeout()** 10-35
virtual memory 1-16, 4-103–4-114

GDB A-4
and logging in as **console** 9-6
compiling for 9-2
debugging loadable kernel servers with 9-7–9-12
enhancements for multithreaded tasks 1-41
running on a panicked system 9-6
gdb NMI mini-monitor command 9-5, 9-12

getbuf functions in network modules 8-4, 8-10, 10-57
GNU source-level debugger *See* GDB

halt NMI mini-monitor command 9-4, 9-6
handler interfaces, setting the function corresponding to a port 10-20
headers, in messages 2-2
HMAP load command 6-9, 9-3, A-5, A-6
host_info() 4-27
host_kernel_version() 4-30
host_priv_self() 4-34
host_processor_set_priv() 4-31
host_processor_sets() 4-32
host_processors() 4-33
host_self() 4-34
hosts
    getting information about 4-27
    getting ports of 4-34
htonl() 10-54
htons() 10-54

ID register, NBIC 7-10
Identification bytes, NeXTbus 7-10
if_attach() 8-3, 8-4, 8-13, 10-54
if_class() 10-56
if_collisions() 10-56
if_collisions_set() 10-56
if_control() 10-57
if_detach() 10-58
if_flags() 10-59
if_flags_set() 10-59
if_getbuf() 10-57
if_handle_input() 10-59
if_ierrors() 10-60
if_ierrors_set() 10-60
if_init() 10-57
if_ioctl() 10-57
if_ipackets() 10-61
if_ipackets_set() 10-61
if_mtu() 10-62
if_name() 10-62
if_oerrors() 10-60
if_oerrors_set() 10-60
if_opackets() 10-61
if_opackets_set() 10-61
if_output() 10-57
if_private() 10-62
if_registervirtual() 8-3, 8-5, 10-63
if_type() 10-62
if_unit() 10-62
iflist_first() 10-64
iflist_next() 10-64
IGNSID0 bit of NBIC Control register 7-5, 7-9
import MiG keyword 2-21
in MiG keyword 2-23
inet_queue() 8-13, 10-64
initialization functions
    in loadable kernel servers 5-5, 6-7
    in loadable kernel servers, debugging 6-7
    in loadable kernel servers, specifying A-5
    in network modules 8-4, 8-6, 10-57
inout MiG keyword 2-23
input functions in network modules 8-4, 8-6
install_polled_intr() 7-7, 10-13
instance variables, in loadable kernel servers 6-6
    specifying A-4
interactive policy 1-25
interprocess communication 1-15
    See also messages
    using Distributed Objects 2-1
    using messages 2-1
    using shared memory 1-12
Interrupt byte, NeXTbus 7-13
interrupt handling 6-3, 7-6
    See also interrupts *and* interrupt level
    installing a handler for a polled device 10-13
    removing a handler 10-53
interrupt level
    getting the current 10-11
    resetting 10-45
    setting 10-43
Interrupt Mask byte, NeXTbus 7-6, 7-13
Interrupt Mask register, NBIC 7-10, 7-13
Interrupt register, NBIC 7-10, 7-13

interrupts
See also interrupt handling *and* interrupt level
at each interrupt level 10-44
generating from the keyboard 9-4
generating non-maskable 9-4
intran MiG keyword 2-19
IP *See* TCP/IP interfaces
IPC *See* interprocess communication

kalloc() 10-14
kattach GDB command 9-6, 9-10
kd_ipaddr NMI mini-monitor command B-6
kern_loader *See* kernel-server loader
kern_loader_abort() 3-3, 4-129
kern_loader_add_server() 3-3, 4-130, 9-3
kern_loader_delete_server() 3-3, 4-131
kern_loader_error() 3-3, 4-133
kern_loader_error_string() 3-3, 4-133
kern_loader_get_log() 3-3, 3-4, 4-134, 6-4, 9-6
kern_loader_load_server() 3-1, 3-2, 3-3, 4-137, 9-3
kern_loader_log_data() 3-9
kern_loader_log_level() 3-3, 4-138, 6-4, 9-6
kern_loader_look_up() 3-3, 4-140
kern_loader_ping() 3-3, 3-4, 3-9, 4-140
kern_loader_reply_handler() 3-4, 3-5, 4-143
kern_loader_reply_t data type 3-5, 3-8, 3-9
kern_loader_server_com_port() 3-3, 4-144
kern_loader_server_info() 3-3, 4-146
kern_loader_server_list() 3-1, 3-3, 4-148
kern_loader_server_task_port() 4-149
kern_loader_status_port() 3-3, 3-4, 3-5, 3-8, 4-150
kern_loader_unload_server() 3-2, 3-3, 4-153
kern_return_t data type 2-10, 2-22
kern_serv_bootstrap_port() 10-14
kern_serv_callout() 10-15
kern_serv_local_port() 10-16
kern_serv_log() 6-4, 10-17
kern_serv_notify() 10-18
kern_serv_notify_port() 10-18
kern_serv_panic() 10-19
kern_serv_port_gone() 10-20
kern_serv_port_proc() 10-20
kern_serv_port_serv() 10-22
kern_serv_port_set() 10-23
kern_serv_unwire_range() 10-24
kern_serv_wire_range() 10-24
kern_server_t data type 6-6
kernel *See* Mach kernel *or* UNIX kernel
KERNEL C preprocessor macro 9-2
kernel ports 1-6
See also task ports *or* thread ports
kernel-server linker *See* **kl_ld** command
kernel-server loader 3-1, 5-4, 9-2, A-1, A-2–A-3
configuration file 9-3, A-2
getting synchronization messages from 4-140
getting the port of 4-140
handling messages from 4-143
monitoring activity of A-8
obtaining the port of 3-3
reconfiguring A-8
requirements of 6-6
shutting down 3-3, A-8
shutting down or reconfiguring 4-129
specifying a port for status messages from 4-150
starting the **kern_loader** daemon A-2
kernel-server loader functions 3-3, 4-129–4-154, A-2
kernel-server log command *See* **kl_log** command
kernel-server utility *See* **kl_util** command
kernel support functions 10-1–10-69
summary C-1–C-8
kernel_thread() 10-25
kernload library 2-11
kfree() 10-26
kget() 6-3, 10-26
kl_ld command 5-5, 9-1, 9-2, 9-3, A-1, A-4–A-7
kl_log command 6-4, 9-6, A-1, A-10
kl_util command 5-4, 9-3, 9-6, 9-10, A-1, A-8–A-9

libc library, thread safety of 1-27
libraries, use of shared data 1-27

load commands script
commands in A-5
example A-7
specifying the file name of A-4
loadable kernel servers
allocated state 3-2, 5-4
allocating 3-3, 4-130, 5-4, 9-3, A-8
building 9-1
choosing an interface for 6-1
concepts behind 3-2
converting to a handler interface 6-8–6-10
deallocating 3-3, 4-131, 5-5, A-8
debugging 9-4
debugging, with GDB 9-7–9-12
designing 6-1
documentation needed for writing 5-6
functions available 6-3
getting communication port of 4-144
getting information about 4-146
getting log data 4-133
getting names of known 4-148
getting status messages from 3-3
getting status of A-8
getting the task port of 4-149
hardware needed for writing 5-6
how they work 5-3
initializing 5-5
instance variable 6-6
loaded state 3-2, 5-4
loading into the kernel 3-3, 4-137, 5-5, 9-3, A-8
and Mach messages 5-4, 6-1, 6-7
names, specifying A-4
network-related *See* network modules
overview of 5-1
process of writing 9-1
program examples 6-7
removing from the kernel 3-3
server vs. handler interfaces 6-7
setting the log level 4-138
states of 3-2, 5-4
tips for writing code 6-2
types available to third parties 5-2
unallocated state 3-2, 5-4
UNIX-style 6-1, 6-13, 5-4, 6-17, 9-3
unloading 4-153, 10-19, A-9
using 3-1
utilities for A-1
loadable object files 5-5, 9-9, 9-10
specifying the file name of A-4
lock_alloc() 10-27
lock_done() 10-28
lock_free() 10-27
lock_init() 10-28
lock_read() 10-29
lock_write() 10-30
locking 1-26
locks
creating in the kernel 10-27
destroying in the kernel 10-27
getting for reading in the kernel 10-29
getting for writing in the kernel 10-30
initializing in the kernel 10-28
releasing in the kernel 10-28
locks, simple
creating in the kernel 10-41
destroying in the kernel 10-41
getting in the kernel 10-40
initializing in the kernel 10-42
releasing in the kernel 10-42
log() 10-30

Mach
compatibility with UNIX 4.3BSD 1-1, 1-3
components of 1-1
concepts behind 1-1
design philosophy 1-2
features not found in UNIX 4.3BSD 1-2
kernel 1-1, 1-4
kernel version, getting 4-30
operating-system support environments 1-1
relation to CMU's Release 2.0 1-1
MACH C preprocessor macro 9-2
Mach communication facility 1-11
See also messages
Mach functions 4-1–4-154
thread safety of 1-26

Mach Interface Generator *See* MiG
Mach kernel functions 4-22–4-114, 6-3
Mach Kit 1-26
Mach messages *See* messages
Mach servers 2-1, 2-8
    common error codes 2-10
    compiling clients 2-11
    creating with MiG 2-9
    programming example 2-11
    registering 2-9
    use by clients 2-10
mach_error() 1-26, 2-10, 4-35, 6-3
mach_error_string() 2-10, 4-35
mach_NeXT_exception() 1-43, 4-36
mach_NeXT_exception_string() 1-43, 4-36
MACH_USER_API C preprocessor macro 9-2
major number 6-13
makefile 6-10
malloc() 1-16, 1-17, 1-26
malloc_singlethreaded() 1-26
map_addr() 7-4, 7-5, 10-31
map_fd() 4-37
master computer 9-7
memory
    See also virtual memory
    allocating 1-16
    allocating in the kernel 10-14, 10-26
    deallocating 1-17
    deallocating in the kernel 10-26
    examining in ROM monitor B-4
    inheritance of 1-14
    mapping files into 1-16, 4-37
    mapping from physical to virtual 7-4
    protection of 1-14, 1-15
    unwiring 10-24
    wiring down 5-5, 10-24, A-7
    zeroing out a region in the kernel 10-7
message body 2-3
message header 2-2
messages 1-8
    access to services through 1-12
    creating by hand 2-1, 2-4
    distinction between Mach and Objective C 1-2
    generating with MiG 2-1
    in loadable kernel servers 6-1
    in a network environment 1-12
    nonsimple 2-6
    receiving 4-39
    reply 2-7
    sending 4-43
    sending and receiving 4-42
    similarity to UNIX 4.3BSD stream sockets 1-12
    simple 2-4, 2-5
    specifying basic data types 2-3
    specifying data size 2-3
    specifying general type 2-2
    specifying meaning of 2-3
    specifying number of data items 2-3
    specifying ports on which to be received or sent 2-2
    specifying size 2-2
    specifying whether data is out-of-line 2-4
    specifying whether large type structure used 2-4
    specifying whether simple 2-2
    specifying whether to deallocate data 2-4
    structure of 2-2
    to a temporarily full port 1-11
    using 2-1
MiG 2-8
    See also **mig** shell command
    data types supported 2-8
    errors 2-11
    in a mixed network environment 2-8
    server vs. handler interfaces 6-8
    specification files 2-9, 2-16
    specification files, compiling 2-30
    specification files, syntax summary 2-27
    type declarations 2-17
    using to create loadable kernel servers 6-1
    using to create Mach servers 2-1, 2-9

mig shell command 2-30, 6-8
See also MiG
mig_symtab_t data type 2-31
migcom shell command 2-32
monitor NMI mini-monitor command 9-4, 9-6
msg NMI mini-monitor command 6-5, 9-5
msg_header_t data type 2-7
msg_id field 2-3
msg_local_port field 2-2, 2-4, 2-7
msg_receive() 1-10, 1-16, 1-39, 2-24, 2-31, 4-39
msg_remote_port field 2-2, 2-4, 2-7
msg_rpc() 2-4, 2-31, 4-42
msg_send() 1-38, 2-4, 2-25, 2-31, 4-43
msg_simple field 2-2, 2-7
msg_size field 2-2
msg_type field 2-2
msg_type_deallocate field 2-4, 2-7
msg_type_inline field 2-4, 2-7
msg_type_long_t data type 2-4
msg_type_longform field 2-4
msg_type_name field 2-3, 2-7
msg_type_number field 2-3, 2-7
msg_type_size field 2-3, 2-7
msg_type_t data type 2-4
MsgError() 2-10, 2-22, 2-26
msgtype MiG keyword 2-23, 2-25
multithreaded applications 1-6, 1-26, 4-2
mutex variables 1-28
associating strings with 4-6
clearing 4-4
creating 4-2
destroying 4-4
initializing 4-5
locking 4-20
unlocking 4-21
use with library calls 1-27
mutex_alloc() 4-2
mutex_clear() 4-4
mutex_free() 4-4
mutex_init() 4-5
mutex_lock() 4-20
mutex_name() 4-6
mutex_set_name() 4-6
mutex_try_lock() 4-20
mutex_unlock() 4-21
mutual exclusion *See* mutex variables

nb_alloc() 10-65
nb_alloc_wrapper() 10-65
nb_free() 10-66
nb_free_wrapper() 10-66
nb_grow_bot() 10-66
nb_grow_top() 10-66
nb_map() 10-67
nb_read() 10-68
nb_shrink_bot() 10-66
nb_shrink_top() 10-66
nb_size() 10-69
nb_write() 10-68
NBIC 5-6, 7-5
registers on CPU board 7-9
netbuf functions 10-65–10-69
summary C-8
netbufs 8-3
changing size of 10-66
creating 10-65
destroying 10-66
getting data from 10-67, 10-68
getting size of data 10-69
writing data to 10-68
netif functions 10-54–10-64
summary C-6
netifs 8-3
detaching 10-58
finding in the system list 10-64
getting information about 10-62
getting or setting flags associated with 10-59
getting the class of 10-56
initializing and installing 10-54
netname_check_in() 4-126
netname_check_out() 4-127
netname_look_up() 2-10, 4-127
network buffers *See* netbufs
network collisions, getting or setting the number of 10-56

network errors
getting or setting number of 10-60
network functions 10-54–10-69
summary C-6
network interfaces *See* netifs
network modules
callback functions 8-4
control functions 8-4, 8-11
device drivers 8-1
Ethernet 8-12
functions implemented in 8-4
getbuf functions 8-4, 8-10
initialization functions 8-4, 8-6
input functions 8-4, 8-6
output functions 8-4, 8-8
overview of 8-2
packet sniffers 8-1
protocol handlers 8-1, 10-59
TCP/IP 8-13
Network Name Server 2-9, 5-4
Network Name Server functions 4-126–4-128
network packets
what happens when received 8-2
what happens when sent 8-2
network packets, getting or setting the number received or sent 10-61
network port identifier 1-12
network ports 1-12
network protocols 5-4
network servers 1-12
networks, using messages in 1-12
NeXTbus
byte ordering 7-6
physical address space 7-2, 7-3
registers 7-8
transferring data 7-5
workaround for intercepted addresses 7-5
writing code to control the hardware 7-4
NeXTbus device drivers 7-1
providing interrupt handlers for 7-6
NeXTbus Interface Chip *See* NBIC
NMI 9-4
how to generate 9-4
NMI mini-monitor 6-5, 9-4, 9-5, B-1
commands B-1, B-6
non-maskable interrupt *See* NMI
nosendtime MiG keyword 2-25
notify ports 1-7
getting a loadable kernel server's 10-18
nowaittime MiG keyword 2-24
ns_abstimeout() 6-3, 10-32
ns_time_to_timeval() 10-34
ns_timeout() 6-3, 10-32
ns_untimeout() 10-35
ntohl() 10-54
ntohs() 10-54

objc_setMultithreaded() 1-27
object files *See* loadable object files *or* relocatable object files
Objective C runtime system, thread safety of 1-27
objects, Mach
communication between 1-8
differences from Objective C objects 1-2
protection of communication between 1-9
out MiG keyword 2-23
out-of-line data 1-17, 2-6, 2-11
in loadable kernel servers 6-10
output functions in network modules 8-4, 8-8, 10-57
outtran MiG keyword 2-19

packet sniffers 8-1
page alignment 1-13, 1-17
page size 1-13
getting 4-113
paging 1-13
object 1-16
Panic window 6-5, 9-4, 9-5, 10-36
panic() 6-5, 10-36
setting a breakpoint for 9-11
panic, kernel 6-5, 6-7, 10-2
See also **panic()** *or* Panic window
physical memory 1-13
ping shell command 9-8
pointer types in MiG 2-21

policies 1-23, 1-25
enabling or disabling on processor sets 4-68
setting for threads 4-93
port access rights 1-9
extracting 4-47
inserting 4-48
passing in messages 1-9
of threads 1-9
port names 1-10
port queues 1-11
setting the size of 4-54
port sets 1-8, 1-10
adding ports to 4-52
creating 4-53
destroying 4-56
getting in loadable kernel servers 10-23
getting the ports in 4-58
removing ports from 4-57
port_all_t data type 1-10
port_allocate() 4-45
port_deallocate() 4-46
PORT_DEATH load command A-6
port_extract_receive() 4-47
port_extract_send() 4-47
port_insert_receive() 4-48
port_insert_send() 4-48
port_name_t data type 1-10
port_names() 4-49
port_rename() 1-10, 4-50
port_set_add() 1-10, 4-52
port_set_allocate() 1-10, 4-53
port_set_backlog() 4-54
port_set_backup() 4-55
port_set_deallocate() 1-10, 4-56
port_set_name_t data type 1-10
port_set_remove() 1-10, 4-57
port_set_status() 1-10, 4-58
port_status() 4-59
port_t data type 1-10
port_type() 4-60
ports 1-8
adding to port sets 4-52
associated with tasks 1-7
associated with tasks, getting 4-76
associated with threads 1-7
associated with threads, getting 4-87
bootstrap 1-7
changing a task's name for 1-10
checking in for services 4-126
creating 4-45
creating for services 4-116
destroying 4-46
determining tasks' rights for 4-60
determining which one the kernel just received a message on 10-16
exception 1-7, 1-36, 1-39
getting for services 4-119, 4-120, 4-127
getting information about a task's 4-49
getting receive rights to 4-115
getting status of 4-59
getting the port used to initialize a loadable kernel server 10-14
kernel 1-6
network 1-12
notify 1-7
notify, of a loadable kernel server 10-18
notifying the kernel about deletions of 10-20
receiving notification messages about in a loadable kernel server 10-18
registering for services 4-121
removing for services 4-127
removing from port sets 4-57
renaming 4-50
reply 1-7
setting the backup 4-55
task 1-6
task, getting the kernel's 10-16
thread 1-6
use of in Mach function calls 1-6
printf() 1-27, 6-4, 6-5, 9-5, 9-6, 10-36

priorities 1-23, 1-24
base 1-24
current 1-24
maximum 1-24
setting for C threads 4-15
setting for tasks 4-80
setting for threads 4-95
setting maximum for processor sets 4-68
probe_rb() 6-5, 10-37
procedure MiG keyword 2-22
processes 1-5
See also tasks
differences from Mach tasks 1-5
processor sets
assigning tasks to 4-73
assigning threads to 4-86
creating 4-64
destroying 4-65
enabling or disabling policies on 4-68
getting information about 4-65
getting ports of 4-31, 4-32, 4-64
getting ports of tasks assigned to 4-70
getting ports of threads assigned to 4-71
setting priorities 1-24, 4-68
processor_assign() 4-61
processor_control() 4-61
processor_exit() 4-61
processor_get_assignment() 4-61
processor_info() 4-62
processor_set_create() 4-64
processor_set_default() 4-64
processor_set_destroy() 4-65
processor_set_info() 4-65
processor_set_max_priority() 4-68
processor_set_policy_disable() 4-68
processor_set_policy_enable() 1-25, 4-68
processor_set_tasks() 4-70
processor_set_threads() 4-71
processor_start() 4-61
processors
controlling 4-61
getting information about 4-62
getting ports of 4-33
programmed I/O 7-5
protocol handlers 8-1
dispatching input packets to 10-59
ps shell command 1-24

quit GDB command 9-12

R ROM monitor command B-5
r ROM monitor command B-3
rcsid MiG keyword 2-27
reboot NMI mini-monitor command 9-4
receive rights 1-9
registers
68040 B-1
NBIC Control 7-5, 7-9
NBIC ID 7-10
NBIC Interrupt 7-13
NBIC Interrupt Mask 7-10, 7-13
NBIC Interrupt register 7-10
NBIC, on CPU board 7-9
NeXTbus 7-8
relocatable object files 5-5, 9-9, A-2, A-4
specifying the file name of A-5
remote procedure calls 2-1
performance implications of 2-8
using MiG to generate code for 2-8
reply ports 1-7
replyport MiG keyword 2-23
requestport MiG keyword 2-23
resetting the CPU 9-4
RMCOL bit of NBIC Control register 7-9
ROM monitor 9-5, 9-6, B-1
commands B-1–B-5
routine MiG keyword 2-22
RPC *See* remote procedure calls
run GDB command 9-11

S ROM monitor command B-5
s ROM monitor command B-3
scheduling
See also priorities *and* policies
of threads 1-23

select() UNIX system call, implementing 6-14, 10-38
selthreadcache() 6-14, 10-38
selthreadclear() 6-14, 10-38
selwakeup() 10-38
send rights 1-9
sendtime MiG keyword 2-23, 2-25
server interfaces, setting the function corresponding to a port 10-22
serverprefix MiG keyword 2-26
servers *See* loadable kernel servers *or* Mach servers *or* network servers
services
  checking availability of 4-123
  creating 4-116
  getting information about 4-117
set_clock() 10-39
shared memory 1-12, 1-14
shared variables 1-27
shutdown functions in loadable kernel servers 5-5, 6-7
  specifying A-7
simple_lock() 10-40
simple_lock_alloc() 10-41
simple_lock_free() 10-41
simple_lock_init() 10-42
simple_unlock() 10-42
simplefunction MiG keyword 2-15
simpleprocedure MiG keyword 2-15, 2-22
simpleroutine MiG keyword 2-22
simport MiG keyword 2-21
skip MiG keyword 2-22, 2-23
slave computer 9-7
slot address space, NeXTbus 7-2
slot driver 5-2
SMAP load command 6-9, 9-3, A-5, A-6
spl*n*() 10-43
splx() 10-45
sprintf() 10-45
standard I/O functions, thread safety of 1-27
START load command 9-3, A-5, A-6
STFWD bit of NBIC Control register 7-9
strcat() 10-46
strcmp() 10-46
strcpy() 10-47
strings, kernel support functions for 10-45, 10-46, 10-47
strlen() 10-47
struct MiG keyword 2-20
subsystem MiG keyword 2-16
superuser, determining whether the user is 10-48
suser() 10-48
syscall() 2-31
system console 9-6
system log, adding to from a loadable kernel server 10-30

task ports 1-6
  getting 4-70
  getting for UNIX processes 4-73
task_assign() 4-73
task_assign_default() 4-73
task_by_unix_pid() 4-73
task_create() 4-74
task_get_assignment() 4-75
task_get_bootstrap_port() 4-77
task_get_exception_port() 1-43, 4-77
task_get_notify_port() 4-77
task_get_special_port() 4-76
task_info() 1-24, 4-78
task_notify() 1-7, 4-76
task_priority() 1-24, 4-80
task_resume() 4-81
task_self() 1-6, 4-76
task_set_bootstrap_port() 4-77
task_set_exception_port() 1-43, 4-77
task_set_notify_port() 4-77
task_set_special_port() 1-7, 4-76
task_suspend() 1-39, 4-82
task_terminate() 1-36, 1-39, 4-82
task_threads() 1-6, 4-83

tasks 1-5
as basic unit of protection 1-5
assigning to processor sets 4-73
corresponding to UNIX processes, getting 4-73
creating 1-14, 4-74
getting current from within the kernel 10-12
getting information about 4-78
getting ports of 4-70, 4-76
getting process ID of 4-102
getting processor set assignment of 4-75
getting threads of 4-83
resuming 4-81
setting ports of 4-76
setting priorities 1-24
setting priority of 4-80
suspending 4-82
terminating 4-82
using Distributed Objects to communicate between 2-1
virtual address space 1-13
TCP/IP interfaces 8-13
queueing input packets 10-64
thread ports 1-6
getting 4-71
thread safety 1-26
thread_abort() 4-84
thread_assign() 4-86
thread_assign_default() 4-86
thread_block() 10-48
thread_create() 4-86
thread_get_assignment() 4-87
thread_get_exception_port() 1-43, 4-88
thread_get_reply_port() 4-88
thread_get_special_port() 4-87
thread_get_state() 1-38, 1-39, 4-89
thread_halt_self() 10-49
thread_info() 1-24, 4-90
thread_max_priority() 1-24, 4-95
thread_policy() 1-25, 4-93
thread_priority() 1-24, 4-95
thread_reply() 1-7, 4-87
thread_resume() 4-98
thread_self() 1-6, 4-87
thread_set_exception_port() 1-7, 1-43, 4-88
thread_set_reply_port() 4-88
thread_set_special_port() 4-87
thread_set_state() 1-38, 1-39, 4-89
thread_set_timeout() 10-49
thread_sleep() 10-50
thread_suspend() 4-99
thread_switch() 1-24, 1-25, 4-100
thread_t data type 4-2
thread_terminate() 1-36, 1-38, 4-101
thread_wait_result() 10-51
thread_wakeup() 10-52
threads 1-5
See also C threads
advantages in multiprocessor architecture 1-6
as basic unit of scheduling 1-5
assigning to processor sets 4-86
causing context switches 4-100
communication between 1-8
creating 4-86
getting and setting the state of 4-89
getting information about 4-90
getting ports of 4-71, 4-87
getting processor set assignment of 4-87
interrupting 4-84
kernel, getting the wait result 10-51
kernel, putting to sleep 10-48, 10-50
kernel, starting new 10-25
kernel, stopping from waiting for an event 10-7
kernel, stopping the current 10-49
kernel, suspending 10-3
kernel, waking up 10-52
multiple, executing within a single task 1-6, 1-26, 4-2
resuming 4-98
scheduling 1-23
setting policies 4-93
setting ports of 4-87
setting priorities 1-24, 4-95
suspending 4-99
terminating 4-101

time
converting between formats 10-34
current, getting or setting from within the kernel 10-9, 10-39
getting information about clocks from within the kernel 10-8
timeout 2-10
timers, in the kernel, setting 10-49
timesharing policy 1-23, 1-25
timeval_to_ns_time() 10-34
traps, functions implemented by 1-8
type MiG keyword 2-17

uimport MiG keyword 2-21
uninstall_polled_intr() 7-7, 10-53
UNIX 4.3BSD
device entry points 6-13
Mach compatibility with 1-1, 1-3
UNIX kernel 1-3
UNIX-style loadable kernel servers 5-4, 6-1, 6-13, 9-3
inserting into device switch tables 6-17
UNIX system calls, thread safety of 1-26
unix_pid() 4-102
unload commands script A-7
example A-7
specifying the file name of A-4
user-level programs, advantages over kernel code 5-1
userprefix MiG keyword 2-26

VALID bit of NeXTbus Identification bytes 7-11
variables, using shared 1-27
virtual address space 1-13, 1-14
virtual memory
See also memory 1-13
allocating 4-103
copying 4-104
deallocating 4-106
getting information about regions of 4-111
getting statistics 4-113
inheriting 4-107
management 1-13
programming examples 1-17
protecting 4-108
reading 4-109
writing 4-114
virtual memory functions 1-16, 4-103–4-114
vm_allocate() 1-16, 1-17, 1-19, 4-103
vm_copy() 1-19, 4-104
vm_deallocate() 1-16, 1-17, 1-19, 2-11, 3-9, 4-106
vm_inherit() 1-20, 1-21, 4-107
vm_page_size global variable 1-13, 4-113
vm_protect() 4-108
vm_read() 1-17, 1-38, 1-39, 4-109, 6-10, 6-11
vm_region() 4-111
vm_stat shell command 9-7
vm_statistics() 4-113
vm_write() 1-38, 1-39, 4-114, 6-10, 6-11
volatile
declaring hardware addresses as 9-2
declaring shared variables as 1-27, 9-2

waittime MiG keyword 2-23, 2-24
Window Server, thread safety of 1-27
WIRE load command A-7